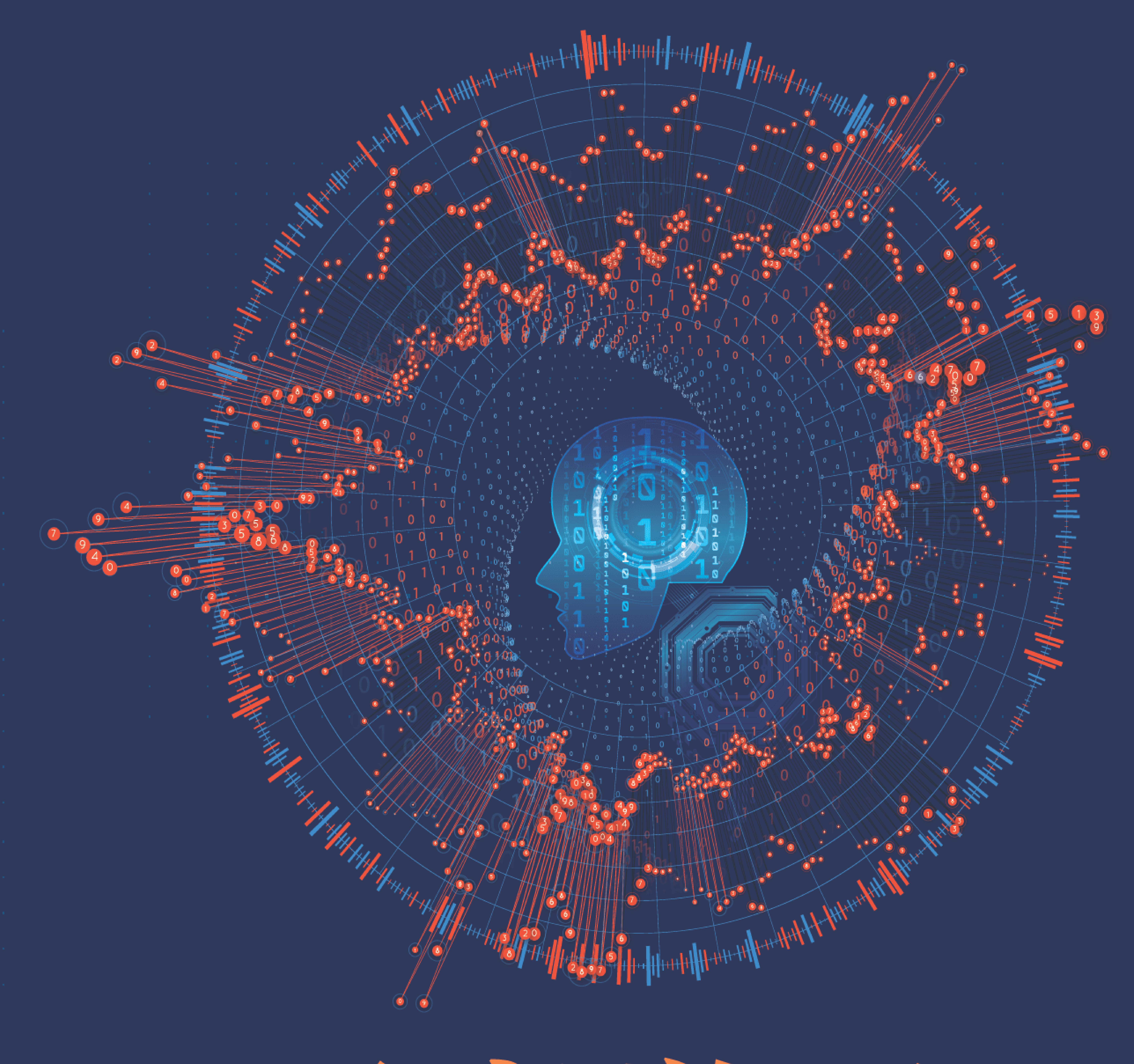


深度学习

语音识别技术实践

- 业内流行的Kaldi语音识别技术实践
- 猎免搜索技术团队语音识别技术总结
- 引领语音识别技术升级

柳若边　编著

清华大学出版社

深度学习：语音识别技术实践

柳若边　编著

清華大學出版社
北　京

内容简介

语音识别已经逐渐进入人们的日常生活。语音识别技术是涉及语言、计算机、数学等领域的交叉学科。本书介绍了包括 C#、Perl、Python、Java 在内的多种编程语言实践，开源语音识别工具包 Kaldi 的使用与代码分析，深度学习的开发环境搭建，卷积神经网络，以及语音识别中常见的语言模型——*N* 元模型和依存模型等，让读者快速了解语音识别基础，掌握开发语音识别程序的算法。

本书从语音识别的基础开始讲起，并辅以翔实的案例，既适合需要具体实现语音识别的程序员使用，也适合有一定机器学习或语音识别基础的学生、研究者或从业者阅读。

图书在版编目（CIP）数据

深度学习：语音识别技术实践 / 柳若边编著. —北京：清华大学出版社，2019

ISBN 978-7-302-51692-7

Ⅰ. ①深…　Ⅱ. ①柳…　Ⅲ. ①机器学习－语音识别－研究　Ⅳ. ①TP181

中国版本图书馆 CIP 数据核字（2018）第 265420 号

责任编辑：张　敏
封面设计：杨玉兰
责任校对：胡伟民
责任印制：

出版发行：清华大学出版社
网　　址：http：//www.tup.com.cn, http：//www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 装 者：
经　　销：全国新华书店
开　　本：186mm×240mm　　**印　　张：**18.25　　**字　　数：**355 千字
版　　次：2019 年 4 月第 1 版　　**印　　次：**2019 年 4 月第 1 次印刷
定　　价：89.00 元

产品编号：078565-01

前　言

作为人工智能技术的重要组成部分，语音识别旨在研究计算机如何听懂人的讲话。来源于人工神经网络的深度学习促进了语音识别技术的发展。本书从使用开源的语音识别构建系统 Kaldi 开始讲起，引导读者亲自实现语音识别系统，使用了 C#、Perl、Python、Java 等多种编程工具。第 1 章介绍语音识别的基本原理和 Kaldi 的基本使用方法，以及使用 Kaldi 开发语音识别系统应用到的 Linux shell 脚本基础；第 2 章介绍使用 C#开发语音识别系统；第 3 章介绍 Perl 语言开发基础；第 4 章介绍开发语音识别系统所需要的 Python 基础；第 5 章介绍使用 Java 开发语音识别系统；第 6 章介绍傅里叶变换、MFCC 特征等常用的语音信号处理方法；第 7 章介绍基本的神经网络和深度学习方法及训练神经网络的反向传播方法；第 8 章介绍语音识别解码阶段用到的语言模型，以及语言模型工具包——KenLM。

本书适合需要具体实现语音识别的程序员使用，对机器学习等相关领域的研究人员也有一定的参考价值。猎兔搜索技术团队已经开发出以本书为基础的专门培训课程和商业软件。

本书由柳若边编著，罗刚、沙芸、张子宪、许想娇、石天盈、张继红、罗庭亮、王全军、刘宇、张天津也参与了本书的部分编创工作。本书相关的参考软件和代码在读者 QQ 群（378025857）的附件中可以找到。Kaldi 及其底层依赖的软件，其复杂程度已经超越了一个人所能掌握的程度。此外，一些具体的细节也可以在读者 QQ 群讨

论。在此，感谢早期合著者、合作伙伴、员工、学员、读者的支持，他们为本书的编创提供了良好的工作基础。技术的融合与创新永无止境，就如同在玻璃容器中水培植物一样，这是一个持久的工作。

编著者

2018 年 12 月

目　录

第 1 章 语音识别技术

语音识别技术，也被称为自动语音识别（Automatic Speech Recognition，ASR），它是一门交叉学科，与人们的生活和学习密切相关。其目标是将说话者的词汇内容转换为计算机可读的输入按键、二进制编码或字符序列等。例如，打银行的客服电话，可以直接和银行系统对话，而不是普通的“请按 1”等把人当成机器的询问。在通信中，可以把对方的语音留言转换成文字，还可以根据识别出的文字识别语义，这样可以让机器和人交流。再如，儿童识别图片后，可以说出这个图中是老虎还是大象，系统使用语音识别技术判断孩子回答是否正确，对于不正确的，系统自动给出提示。

做好开放式语音识别不容易，可以辅助人工输入字幕，类似于语音输入法。

1.1 总体结构

语音识别可以看成是广义上的标注问题。给定声学输出 $A_{1,T}$（由一个声学事件的序列组成 $a_1,\cdots,a_T$），需要找到单词序列 $W_{1,R}$ 的最大化概率：

$$\arg\max_{\omega} P\left(W_{1,R} \mid A_{1,T}\right)$$

根据贝叶斯公式重写上述公式，并删除在通过比较大小找最大值的过程中没有意义的分母，把问题转换成计算：

$$\arg\max_{\omega} P\left(A_{1,T} \mid W_{1,R}\right) P\left(W_{1,R}\right)$$

这里将 $P(A_{1,T} \mid W_{1,R})$称为声学模型，而将 $P(W_{1,R})$称为语言模型。语言识别结构如图 1-1 所示。

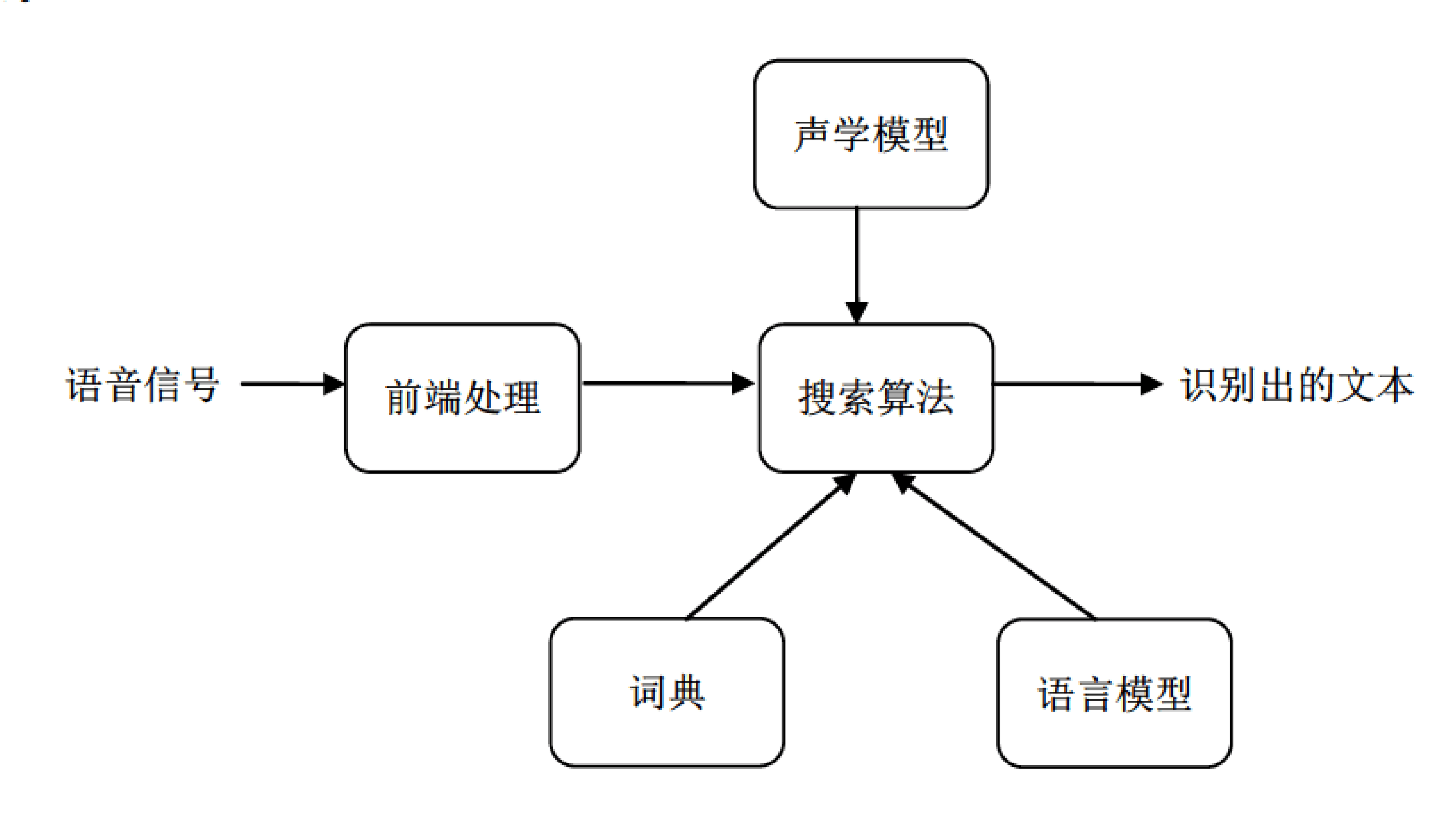


图 1-1　语音识别结构

人类获得信息的 80% 都来自图像。图像信息具有传递速度快、信息量大等一系列特点，因此图像信息得到了广泛的应用。但语音识别在车载系统、智能音响等领域也有非常关键的应用。

为了能开发出有效的语音识别系统，2009 年 Kaldi 在约翰 • 霍普金斯大学诞生了。Kaldi 不是一款语音识别系统，而是一款建立语音识别系统的系统。Kaldi 使用运行于 Linux 操作系统的 C++、Perl、Python、Bash 等多种语言开发。接下来介绍需要用到的 Linux 基础知识。

1.2　Linux 基础

Linux 是围绕 Linux 内核构建的免费和开源软件操作系统系列。通常，Linux 以桌面和服务器使用的称为 Linux 发行版的形式打包。Linux 有一些常用的发行版 CentOS 和 Ubuntu 等版本。

和 CentOS 不同，Ubuntu 操作系统上没有管理员知道 root 密码（root 密码是随机生成的），而 root 权限是通过操作 sudo 命令授予的。

有些语音识别系统运行在 Linux 服务器中，为了远程登录 Linux 服务器，用户可以先在 Windows 下安装 Chrome 浏览器，然后可以通过网址 http://sshy.us/登录 Linux 服务器。在界面中输入 IP 地址、用户名和密码，如果是用 root 账户登录，则终端提示符为“#”；否则，终端提示符为“$”。

查看 Ubuntu 操作系统版本号：

```
$cat /etc/issue
Ubuntu 18.04 LTS \n \l
```

或者：

```
$lsb_release -r
Release:        18.04
```

获取 Ubuntu 的代号：

```
$lsb_release -c
Codename:       bionic
```

ls 命令用于列出当前目录下的文件；history 命令用于显示历史。有的命令比较长，为了实现快速输入，可以用 Tab 键补全命令；也可以用上箭头选择最近运行过的命令再次执行。

连接到远程 Linux 服务器可以使用支持 SSH 协议的终端仿真程序 SecureCRT。因为它可以保存登录密码，所以应用起来比较方便。除了 SecureCRT，还可以使用开源软件 PuTTY（http://www.chiark.greenend.org.uk/~sgtatham/putty），以及可以保存登录密码的 PuTTY Connection Manager。在终端启动的进程断开连接后会停止运行。为了让进程继续运行，可以使用 nohup 命令。

如果需要安装软件，可以下载对应的 RPM 安装包，然后使用 RPM 安装。但操作系统对应的 RPM 安装包找起来往往比较麻烦——一个软件包可能依赖其他的软件包；为了安装一个软件，可能需要下载其他的多个它所依赖的软件包。

为了简化安装操作步骤，可以使用黄狗升级管理器（Yellow dog Updater, Modified），

一般简称 YUM。YUM 会自动计算出程序之间的相互关联性，并且计算出完成软件包的安装需要哪些步骤。这样在安装软件时，不会再被那些关联性问题所困扰。

YUM 软件包管理器会自动从网络下载并安装软件。YUM 有点类似 360 软件管家，但是不会有商业倾向的推销软件。例如安装支持 wget 和 lrzsz 命令的软件，执行以下命令行：

```
#yum install wget
#yum install lrzsz
```

Windows 格式文本文件的换行符为\r\n，而 Linux 文件的换行符为\n。dos2unix 命令是将 Windows 格式文件转换为 Linux 格式的实用命令，其实就是将文件中的\r\n 转换为\n。

开发语音识别系统的过程中，可能会用到大量的数据文件。例如需要在 Linux 操作系统上维护同一文件的两份或多份副本，除了保存多份单独的物理文件副本以外，还可以采用保存一份物理文件副本和多个虚拟副本的方法。这种虚拟的副本就称为链接。链接是目录中指向文件真实位置的占位符。在 Linux 中有两种不同类型的文件链接——符号链接和硬链接。其中，符号链接是一个实实在在的文件，它指向存放在虚拟目录结构中某个地方的另一个文件。这两个通过符号链接在一起的文件，彼此的内容并不相同。

对于过大的文件，可以使用 wget 命令在后台下载。

```
#wget -bc <path>
```

这里的参数 b 表示在后台运行；参数 c 表示支持断点续传。

1.3 安装 Micro 编辑器

为了方便在服务器端开发 Python\Perl 的相关应用，可以采用 Micro（https://github.com/zyedidia/micro）这样的终端文本编辑器。如果有 Snap 安装工具软件，可以使用 Snap

安装 Micro：

```
#snap install micro --classic
```

如果没有 Snap 安装工具软件，也可以直接安装 Micro 的预编译版本：

```
#wget  https://github.com/zyedidia/micro/releases/download/nightly/micro-1.3.4-67-linux64.tar.gz
#tar -xf ./micro-1.3.4-67-linux64.tar.gz
```

编辑/etc/profile 文件，增加 micro 所在的路径/home/soft/ micro-1.3.4-67 到 PATH 环境变量下。

```
#./micro /etc/profile
export PATH=/home/soft/micro-1.3.4-67:$PATH
```

可以使用它编辑配置文件：

```
#./micro  run.pl
```

输入以下命令行：

```
die "run.pl: Hello Error";
```

这里的 die 表示终止脚本运行，并显示出 die 后面双引号中的内容。

保存文件后，按 Ctrl+Q 组合键退出。

1.4 安装 Kaldi

一般在 Linux 操作系统下运行 Kaldi，下面讲解在 CentOS 下安装 Kaldi。

首先安装 Git。

```
#yum install git
```

然后下载 Kaldi。

```
#git clone https://github.com/kaldi-asr/kaldi.git kaldi --origin upstream
```

可以参考下载文件中的说明安装。在源码的根目录下有一个 INSTALL 文件，其中描

述了安装步骤。在 tools/下查看 INSTALL 安装指令，然后在 src/下查看 INSTALL 安装指令。到/tools/extras 目录下，运行 check_dependencies.sh 脚本，检查安装过程中所依赖的工具是否存在，运行的结果会提示安装依赖软件的命令。在./tool 目录下编译源代码，然后在./src 目录下编译。

在./tool 目录下只需输入 make 命令就可以编译，输入 make -j 4 命令可以用多核并行处理的方式加快速度。

切换到./src 目录下，运行如下命令：

```
./configure
make depend
make -j 4
```

为了方便以后使用，可以把环境打包：

```
tar -czf kaldiLinux.tar.gz ./kaldi
```

egs 目录下保存着一些指定数据集上的训练步骤（shell 脚本）及测试的结果。最简单的是 yesno 例子。

1.5 yesno 例子

首先运行以下这个例子。

```
#cd ./egs/yesno/s5
#./run.sh
```

经过一段时间的训练和测试，可以看到以下运行结果：

```
%WER 0.00 [ 0 / 232, 0 ins, 0 del, 0 sub ] exp/mono0a/decode_test_yesno/wer_10
```

这里的 WER（Word Error Rate）是字错误率，是一个衡量语音识别系统准确程度的度量。其计算公式为 $WER=(I+D+S)/N$，其中 I 代表被插入的单词个数；D 代表被删除的单词个数；S 代表被替换的单词个数。也就是说，把识别出来的结果中，多认的、少认的和认错的全都加起来，再除以总单词数。这个数值当然是越低越好。这里的 *WER*

为 0.00，说明全部识别正确。

数据集有 62 个.wav 文件，采样频率为 8kHz。所有音频文件由 Kaldi 项目的匿名男性贡献者记录，并包含在项目中用于测试目的。把它们放在 wave_yesno 目录中，但数据集也可以在 http://openslr.org/resources/1/waves_yesno.tar.gz 中找到。在每个文件中，这个人说 8 个字，每个单词都是“ken”或“lo”（希伯来语中的“是”和“否”），因此每个文件都是 8 个“是”或“否”的随机序列。以下文件名称用单词序列表示，1 代表“是”，0 代表“否”。

```
waves yesno/1 0 1 1 1 0 1 0.wav
waves_yesno/0_1_1_0_0_1_1_0.wav
...
```

1.5.1 数据准备

将 62 个波形文件分为两半：31 个用于训练，其余用于测试。创建数据目录，在其中创建两个子目录 train_yesno 和 test_yesno。使用一个命名为 data_prep.py 的 Python 脚本生成必要的输入文件。读取 wave_yesno 中的文件列表。生成两个列表，一个存储以 0 开头的文件名称，另一个存储以 1 开头的名称，忽略其余的文件。

对于每个数据集（训练集和测试集）都需要生成代表原始数据的文件——音频文件和讲稿文件。

1. 讲稿文件

对于讲稿文件，每行一句话，语法格式为：

```
<utt_id> <transcript>
```

例如： 0_0_1_1_1_1_0_0 NO NO YES YES YES YES NO NO

这里使用没有扩展名的文件名作为 utt_ids。虽然录音语言是希伯来语，但是这里使用英语单词 YES 和 NO 代替，以免使问题复杂化。

2. wav.scp

对于唯一 ID 的索引文件，语法格式为：

```
<file_id> <带路径的 wave 文件名或者获取.wav 文件的命令>
```

例如： 0_1_0_0_1_0_1_1 waves_yesno/0_1_0_0_1_0_1_1.wav

这里再次使用文件名作为文件 ID。

3. utt2spk

对于每句话，标记哪个说话者说出来，语法格式为：

```
<utt_id> <speaker_id>
```

例如：0_0_1_0_1_0_1_1 global

由于这个例子中只有一名说话者，所以使用“global”作为说话者标识。

4. spk2utt

对于简单的反向索引，可以使用 Kaldi 工具程序生成：

```
utils/utt2spk_to_spk2utt.pl data/train_yesno/utt2spk > data/train_yesno/spk2utt
```

最后数据目录看起来像这样：



```
data
├───train_yesno
│    ├───text
│    ├───utt2spk
│    ├───spk2utt
│    └───wav.scp
└───test_yesno
     ├───text
     ├───utt2spk
     ├───spk2utt
     └───wav.scp
```

1.5.2 词典准备

本小节讲解如何为 Kaldi 识别器构建语言知识——词典和音素词典。

接下来建立词典。先从根目录创建中间的 dict 目录开始。

```
mkdir dict
```

在这种语言中，只有 YES 和 NO 两个词。为了简单起见，假设它们是单音素词 Y 和 N。

```
echo -e "Y\nN" > dict/phones.txt           #phones dictionary
echo -e "YES Y\nNO N" > dict/lexicon.txt   #word-pronunciation dictionary
```

然而，在真实的讲话中，不仅有表达语言的人的声音，还有沉默和噪声。Kaldi 把所有这些非语言的部分称为“沉默”。例如，即使在这个小而受控制的录音中，也会在每个单词之间暂停。因此，需要一个额外的音素“SIL”代表沉默，而且可以在所有单词的结尾发生。Kaldi 中称这种沉默为“可选的沉默”。

```
echo "SIL" > dict/silence_phones.txt
echo "SIL" > dict/optional_silence.txt
mv dict/phones.txt dict/nonsilence_phones.txt
```

修改词典以包含沉默：

```
cp dict/lexicon.txt dict/lexicon_words.txt
echo "<SIL> SIL" >> dict/lexicon.txt
```

注意，“<SIL>”也将用作 OOV 词。

dict 目录下应该最终得到以下这 5 个文件。

- lexicon.txt：词素-音素对的完整列表。
- lexicon_words.txt：单词-音素对列表。
- silence_phones.txt：无声音素列表。
- nonsilence_phones.txt：非无声音素列表。
- optional_silence.txt：可选无声音素列表（看起来和 silence_phones.txt 相同）。

最后，需要将字典转换为 Kaldi 接受的数据结构——有限状态转换（FST）。在 Kaldi 提供的许多脚本中，将使用 utils/prepare_lang.sh 生成 FST 就绪的数据格式来表示语言定义。

```
utils/prepare_lang.sh --position-dependent-phones false <RAW_DICT_PATH>
<OOV> <TEMP_DIR> <OUTPUT_DIR>
```

其中，--position-dependent-phones 参数值为假，因为这里没有足够的上下文。对于所需参数，我们将使用

- <RAW_DICT_PATH>：dict。
- <OOV>："<SIL>"。
- <TEMP_DIR>：可以在任何地方。这里在 dict 中创建一个新的目录 tmp。
- <OUTPUT_DIR>：此输出将用于进一步的训练，将其设置为 data/lang。

给出了一个用于 yesno 数据的样例一元语言模型。会在 lm 目录下找到一个 arpa 格式的语言模型，然而语言模型也需要转换成 FST。为此，Kaldi 也带来了许多程序。在这个例子中，将使用绑定的脚本 lm/prepare_lm.sh，它将生成正确格式的 LM FST 并将其放入 data/lang_test_tg 中。

接下来是 MFCC 特征提取和训练 GMM 模型。

首先提取梅尔频率倒谱系数。

```
steps/make_mfcc.sh --nj <N> <INPUT_DIR> <OUTPUT_DIR>
```

- --nj <N>：处理器数量，默认为 4。
- <INPUT_DIR>：放训练集数据的地方。
- <OUTPUT_DIR>：允许输出到 exp/make_mfcc/train_yesno，遵循 Kaldi 配方惯例。

现在归一化倒谱特征。

```
steps/compute_cmvn_stats.sh <INPUT_DIR> <OUTPUT_DIR>
```

<INPUT_DIR>和<OUTPUT_DIR>与上述相同。这些 shell 脚本（.sh）都是通过 Kaldi 二进制文件的管道操作，它们都是些文本处理操作。要查看实际执行了哪些命令，请参阅<OUTPUT_DIR>中的日志文件，或者最好查看脚本内容。

训练单音素模型。因为假设在该语言中，音素不依赖于上下文。

```
steps/train_mono.sh --nj <N> --cmd <MAIN_CMD> <DATA_DIR> <LANG_DIR> <OUTPUT_DIR>
```

- --cmd <MAIN_CMD>：因为要使用本地机器资源，所以使用 utils/run.pl 管道。
- --nj <N>：来自说话者的发言不能并行处理。由于只有一个，因此只能使用 1

个任务。

- <DATA_DIR>：训练数据的路径。
- <LANG_DIR>：语言定义的路径（prepare_lang 脚本的输出）。
- <OUTPUT_DIR>：和前面一样，使用 exp/mono。

这将产生用于声学模型的基于 FST 的词图。

```
/path/to/kaldi/src/fstbin/fstcopy'ark:gunzip -c exp/mono/fsts.1.gz|'ark,t:- | head -n 20
```

上述命令，将以人可读的格式打印出前 20 行词图（每列表示：Q-from，Q-to，S-in，S-out，Cost）。

图解码。对于解码，需要一个新的输入，可以通过 AM&LM 词图实现。此外，为 data/test_yesno 准备了单独的测试集。然后，需要建立一个完全连接的 FST 网络。

```
utils/mkgraph.sh --mono data/lang_test_tg exp/mono exp/mono/graph_tgpr
```

将在 exp/mono/graph_tgpr 目录中构建一个连接的 HCLG。

最后，需要使用解码脚本找到测试集中话语的最佳路径。查看解码脚本，找出什么可以作为它的参数，然后运行它。将解码结果写入 exp/mono/decode_test_yesno。

```
steps/decode.sh
```

上述命令将产生输出目录中的 lat.N.gz 文件，其中 *N* 从 1 增加到使用的作业数（对这个任务来说，必须为 1）。这些文件包含由解码操作的第 *N* 个线程处理的发言词图。请参阅 exp/mono/decode_test_yesno/wer_X 文件以查看 WER，参阅 exp/mono/ decode_test_yesno/scoring/X.tra 查看讲稿。这里 X 表示语言模型权重（LMWT），每次迭代使用的评分脚本，将 lat.N.gz 文件中的话语的最佳路径解释为单词序列（记住 N 在 decoing 操作期间#thread）。如果需要，可以在调用 score.sh 时使用--min_lmwt 和--max_lmwt 选项来特意指定权重。如果有兴趣获取每个重新编码文件的单词级对齐信息，请查看 steps/get_ ctm.sh 脚本。

1.6 构建一个简单的 ASR

本节将数据分为训练和测试集，建立一个 ASR 系统，对其进行训练、测试并获得一些解码结果。在 kaldi-trunk/egs 目录中创建一个 digits 文件夹，把所有与项目相关的文件等都放在这里。

假设想要建立一个基于自己的音频数据的 ASR 系统，需要 100 个文件的数据集，文件格式为.wav，每个文件包含 3 个以英文记录的口语数字。这些音频文件中的每一个以可识别的方式命名（例如，“1_5_6.wav”，相应的语音句子是“一、五、六”），并且在特定记录会话期间放置在表示特定说话者可识别的文件夹中。可能有一种情况——同一人的录音在两个不同的质量/噪声环境中，这时要将它们放在单独的文件夹中。总结一下，示范数据集如下所述：

- 10 个不同的说话者（ASR 系统必须在不同的说话者上进行训练和测试，所以说话者越多越好）。
- 每位说话者说 10 句话。
- 100 个句子/话语（100 个*.wav 文件中放置在与特定说话者有关的 10 个文件夹中，每个文件夹中有 10 个* .wav 文件）。
- 300 个词（0～9 的数字）。
- 每个句子/话语由 3 个词组成。

无论第一个数据集是什么，请根据具体情况调整。小心大数据集和复杂的语法，从简单的开始，在这种情况下只包含数字的句子比较不错。

找到 kaldi-trunk/egs/digits 目录并创建 digits_audio 文件夹，在 kaldi-trunk/egs/digits/digits_audio 中再创建两个文件夹——test 和 train。选择一个说话者来表示测试数据集，使用该说话者的“speakerID”作为 kaldi-trunk/egs/digits/digits_audio/test 目录中另一个新文件夹的名称，然后把所有与该人有关的音频文件放在一起；将其余（9 个说话者）放入 train 文件夹，这将是训练数据集。

1．音频数据

准备音频数据。必须创建一些文本文件，用来允许 Kaldi 系统与音频数据打交道。在此部分（及语言数据部分）创建的每个文件都可以被视为具有一定数量的字符串（每个字符串占一个独立行）的文本文件。这些字符串需要排序，如果遇到任何排序问题，可以使用 Kaldi 脚本检查（utils/validate_ data_dir.sh）和修复（utils/fix_data_dir.sh）数据顺序。并且为了信息，utils 目录将在工具附件部分被附加到用户的项目。

在 kaldi-trunk/egs/digits 目录中创建 test 和 train 子文件夹后，在每个子文件夹中创建以下文件（在 test 和 train 子文件夹中以同样的方式命名文件，只不过是与之前创建的两个不同的数据集相关）。

（1）spk2gender

该文件记录说话者的性别。如同我们假设的，“speakerID”是每个说话者的唯一名称（在这种情况下，它也是一个“recordingID”）。在这个例子中，有 5 名女性和 5 名男性说话者（f=女性，m=男性）。

模式：<speakerID> <gender>

例如：

```
cristine f
dad m
josh m
july f
#...
```

（2）wav.scp

该文件用相关的音频文件连接每个发言（在特定记录会话期间由一个人说的句子）。如果坚持这里的命名方法，utteranceID 只不过是 speakerID（说话者的文件夹名称）附带了*.wav 文件名，而没有'.wav'结尾（看下面的例子）。

模式：<uterranceID> <full_path_to_audio_file>

例如：

```
dad_4_4_2 /home/{user}/kaldi-trunk/egs/digits/digits_audio/train/dad/4_4_2.wav
```

```
july 1 2 5 /home/{user}/kaldi-trunk/egs/digits/digits audio/train/july/1 2 5.wav
july_6_8_3 /home/{user}/kaldi-trunk/egs/digits/digits_audio/train/july/6_8_3.wav
#...
```

（3）text

该文件包含与其文本转录匹配的每个发言。

模式：<uterranceID> <text_transcription>

例如：

```
dad 4 4 2 four four two
july 1 2 5 one two five
july 6 8 3 six eight three
#...
```

（4）utt2spk

该文件告诉 ASR 系统发言属于哪个特定的说话者。

模式：<uterranceID> <speakerID>

例如：

```
dad 4 4 2 dad
july 1 2 5 july
july 6 8 3 july
#...
```

（5）corpus.txt

该文件目录略有不同。 在 kaldi-trunk/egs/digits/data 中创建一个文件夹 local，在 kaldi-trunk/egs/digits/data/local 中创建一个文件 corpus.txt，该文件应该包含可能发生在 ASR 系统中的每一个话语转录（在我们的情况下，是 100 行 100 个音频文件）。

模式：<text_transcription>

例如：

```
one two five
six eight three
four four two
#...
```

2. 语言数据

下面涉及的语言建模文件，也需要将其视为“必须完成”。现在展示的是一个理想的例子。

在 kaldi-trunk/egs/digits/data/local 目录中创建一个文件夹 dict，在 kaldi-trunk/egs/digits/data/local/dict 中创建以下文件。

（1）lexicon.txt

该文件包含字典中的每个单词及其电话录音（取自/egs/voxforge）。

模式：<word> <phone 1> <phone 2>…

例如：

```
!SIL sil
<UNK> spn
eight ey t
five f ay v
four f ao r
nine n ay n
one hh w ah n
one w ah n
seven s eh v ah n
six s ih k s
three th r iy
two t uw
zero z ih r ow
zero z iy r ow
```

（2）nonsilence_phones.txt

该文件列出了项目中存在的非静止的音素。

模式：<phone>

```
ah
ao
ay
eh
ey
```

```
f
hh
ih
iy
k
n
ow
r
s
t
th
uw
w
v
z
```

（3）silence_phones.txt

该文件列出了静止的音素。

模式：<phone>

```
sil
spn
```

（4）optional_silence.txt

该文件列出了可选的静止音素。

模式：<phone>

```
sil
```

下面添加在示例性脚本中广泛使用的必需的 Kaldi 工具。

从 kaldi-trunk/egs/wsj/s5 目录中复制两个文件夹（和全部内容）——utils 和 steps，并将它们放在 kaldi-trunk/egs/digits 目录中。同时，还可以创建到这些目录的链接，可以在 kaldi-trunk/egs/voxforge/s5 中找到这样的链接。

打分脚本可帮助获得解码结果，从 kaldi-trunk/egs/voxforge/s5/local 目录中将脚本 score.sh 复制到项目中的相似位置（kaldi-trunk/egs/digits/local）。此外，还需要安装在这个示例中使用的语言建模工具包——SRI 语言建模（SRILM）工具包。有关详细的安

装说明，请访问 kaldi-trunk/tools/install_srilm.sh（阅读里面所有的注释）。因为 SRILM 是一款商用收费的软件，所以没有自动下载的脚本。下载 srilm-1.7.2.tar.gz 并重命名为 srilm.tgz 后，运行 install_srilm.sh。

```
mv ./srilm-1.7.2.tar.gz  ./srilm.tgz
./install_srilm.sh
```

在 kaldi-trunk/egs/digits 中创建一个文件夹 conf，在该文件夹内部创建以下两个文件（对于解码和 mfcc 特征提取过程中的一些配置修改，取自/egs/voxforge）。

（1）decode.config

```
first beam=10.0
beam=13.0
lattice_beam=6.0
```

（2）mfcc.conf

```
--use-energy=false
```

最后一项工作是准备运行脚本来创建 ASR 系统。

以下这两种方法足以在仅使用数字词典和小训练数据集的情况下显示解码结果的显著差异。

- MONO：单声道训练。
- TRI1：简单的三音素训练。

在 kaldi-trunk/egs/digits 目录中创建以下 3 个脚本。

（1）cmd.sh

```
#设置本地系统任务(本地 CPU 无须外部群集)
export train cmd=run.pl
export decode_cmd=run.pl
```

（2）path.sh

```
#定义 Kaldi 根目录
export KALDI ROOT='pwd'/../..
#设置路径以包含有用的工具
export PATH=$PWD/utils/:$KALDI_ROOT/src/bin:$KALDI_ROOT/tools/openfst/
bin:$KALDI_ROOT/src/fstbin/:$KALDI_ROOT/src/gmmbin/:$KALDI_ROOT/src/featbin/
```

```
:$KALDI_ROOT/src/lmbin/:$KALDI_ROOT/src/sgmm2bin/:$KALDI_ROOT/src/fgmmbin/
:$KALDI_ROOT/src/latbin/:$PWD:$PATH
    #定义音频数据路径
    export DATA_ROOT="/home/{user}/kaldi-trunk/egs/digits/digits_audio"
    #启用 SRILM
    source $KALDI_ROOT/tools/env.sh
    #为了数据能够正确排序所需的变量
    export LC_ALL=C
```

（3）run.sh

```
    #!/bin/bash
    ../path.sh || exit 1
    ../cmd.sh || exit 1
    nj=1          #并行任务的数量。对于这样一个小数据集，取 1 就很好
    lm_order=1    #语言模型阶数（n-gram 数量）。对于数字文法来说，取 1 是够用的
    #安全机制(可能使用修改后的参数运行此脚本)
    . utils/parse_options.sh || exit 1
    [[ $#-ge 1 ]] && { echo "Wrong arguments!"; exit 1; }
    #删除以前创建的数据(从上次的 run.sh 执行)
    rm -rf exp mfcc data/train/spk2utt data/train/cmvn.scp data/train/feats.scp
data/train/split1 data/test/spk2utt data/test/cmvn.scp data/test/feats.scp data/
test/split1 data/local/lang data/lang data/local/tmp data/local/dict/lexiconp.txt
    echo
    echo "===== PREPARING ACOUSTIC DATA ====="
    echo
    #需要手工准备(或使用自己写的脚本)
    #
    #spk2gender      [<speaker-id> <gender>]
    #wav.scp         [<uterranceID> <full_path_to_audio_file>]
    #text            [<uterranceID> <text_transcription>]
    #utt2spk         [<uterranceID> <speakerID>]
    #corpus.txt      [<text_transcription>]
    #制作 spk2utt 文件
    utils/utt2spk_to_spk2utt.pl data/train/utt2spk > data/train/spk2utt
    utils/utt2spk_to_spk2utt.pl data/test/utt2spk > data/test/spk2utt
    echo
    echo "===== FEATURES EXTRACTION ====="
```

```
    echo
    #制作 feats.scp 文件
    mfccdir=mfcc
    #如果在数据排序方面遇到任何问题,就在下面的脚本中取消注释并修改参数
    #utils/validate_data_dir.sh data/train  #检查准备好数据的脚本，在这里路径为
data/train
    #utils/fix_data_dir.sh data/train #数据正确排序的工具，在这里路径为 data/train
    steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/train exp/make_mfcc/
train $mfccdir
    steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/test exp/make_mfcc/test
$mfccdir

    #制作 cmvn.scp 文件
    steps/compute_cmvn_stats.sh data/train exp/make_mfcc/train $mfccdir
    steps/compute_cmvn_stats.sh data/test exp/make_mfcc/test $mfccdir
    echo
    echo "===== PREPARING LANGUAGE DATA ====="
    echo
    #需要手工准备(或使用自己写的脚本)
    #
    #lexicon.txt              [<word> <phone 1> <phone 2> ...]
    #nonsilence phones.txt    [<phone>]
    #silence phones.txt       [<phone>]
    #optional silence.txt     [<phone>]

    #准备语言数据
    utils/prepare_lang.sh data/local/dict "<UNK>" data/local/lang data/lang
    echo
    echo "===== LANGUAGE MODEL CREATION ====="
    echo "===== MAKING lm.arpa ====="
    echo
    loc='which ngram-count';
    if [ -z $loc ]; then
         if uname -a | grep 64 >/dev/null; then
              sdir=$KALDI_ROOT/tools/srilm/bin/i686-m64
         else
              sdir=$KALDI_ROOT/tools/srilm/bin/i686
```

```
        fi
        if [ -f $sdir/ngram-count ]; then
            echo "Using SRILM language modelling tool from $sdir"
            export PATH=$PATH:$sdir
        else
            echo "SRILM toolkit is probably not installed.Instructions:
                  tools/install_srilm.sh"
            exit 1
        fi
    fi

    local=data/local
    mkdir $local/tmp
    ngram-count-order $lm_order-write-vocab $local/tmp/vocab-full.txt-wbdiscount
-text $local/corpus.txt -lm $local/tmp/lm.arpa
    echo
    echo "===== MAKING G.fst ====="
    echo
    lang=data/lang
    arpa2fst --disambig-symbol=#0 --read-symbol-table=$lang/words.txt $local/
tmp/lm.arpa $lang/G.fst
    echo
    echo "===== MONO TRAINING ====="
    echo
    steps/train_mono.sh --nj $nj --cmd "$train_cmd" data/train data/lang exp/
mono || exit 1
    echo
    echo "===== MONO DECODING ====="
    echo
    utils/mkgraph.sh --mono data/lang exp/mono exp/mono/graph || exit 1
    steps/decode.sh --config conf/decode.config --nj $nj --cmd "$decode_cmd"
exp/mono/graph data/test exp/mono/decode
    echo
    echo "===== MONO ALIGNMENT ====="
    echo
    steps/align_si.sh --nj $nj --cmd "$train_cmd" data/train data/lang exp/mono
exp/mono ali || exit 1
    echo
```

```
    echo "===== TRI1 (first triphone pass) TRAINING ====="
    echo
    steps/train_deltas.sh --cmd "$train_cmd" 2000 11000 data/train data/lang
exp/mono ali exp/tri1 || exit 1
    echo
    echo "===== TRI1 (first triphone pass) DECODING ====="
    echo
    utils/mkgraph.sh data/lang exp/tri1 exp/tri1/graph || exit 1
    steps/decode.sh --config conf/decode.config --nj $nj --cmd "$decode_cmd"
exp/tri1/graph data/test exp/tri1/decode
    echo
    echo "===== run.sh script is finished ====="
    echo
```

现在要做的就是运行 run.sh 脚本。终端的日志能提示你如何处理可能遇到的错误。

除了在终端窗口中会注意到某些解码结果外，进入新创建的 kaldi-trunk/egs/digits/exp，可能会注意到该 exp 文件夹中有同样目录结构的 mono 和 tri1 结果。如果切换到 mono/decode 目录，在这里可能会找到结果文件（以 wer_ {number}方式命名）。解码过程的日志可以在日志文件夹（同一目录）中找到。

在成功安装 Kaldi 后，可运行一些示例脚本（如 Yesno、Voxforge、LibriSpeech，它们相对容易，有免费的音频/语言数据可供下载）。

1.7 Voxforge 例子

将./path.sh 中的变量 DATA_ROOT 设置为指向数据驻留的目录。使用./getdata.sh 下载 VoxForge 的压缩数据至${DATA_ROOT}/ tgz，并提取这些数据到${DATA_ROOT}/extracted。

所使用的主要子目录包括以下几个。

- local/：存放每个例子特有的脚本。这主要是数据归一化的内容——从具体的语音数据库获取信息，并将其转换为后续期望的文件/数据结构。用 Kaldi 使用新

数据的工作会涉及编写和修改该类别的脚本。

- conf/：存放小的配置文件，指定如特征提取参数和解码束。
- steps：存放实现各种声学模型训练方法的脚本，主要通过调用 Kaldi 的二进制工具和 utils/中的脚本。
- utils/：存放执行小型底层任务的脚本，例如向词典添加消除歧义符号，在单词的符号和整数表示之间进行转换。
- data/：存储配方运行时生成的各种元数据，大部分是由 local/中的脚本生成的。
- exp/：配方最重要的输出（包括声学模型和识别结果）就存放在这里。
- tools/：存放各种外部工具。

该配方可以选择对 VoxForge 数据的一部分进行训练和测试。对于（几乎）每个提交目录都有一个 etc/README 文件，带有发音方言的元信息和说话者的性别。例如，选择如下四国的英语：

```
dialects="((American)|(British)|(Australia)|(Zealand))"
```

如果想选择所有 VoxForge 的英文演讲，应该将其设置为：

```
dialects="English"
```

VoxForge 允许匿名演讲者注册和演讲，从 Kaldi 的脚本观点来看，这样不理想，因为 Kaldi 要执行说话者依赖变换。“匿名”讲话记录在不同环境/通道条件（使用麦克风、背景噪声等）下，演讲者可能是男性，也可能是女性，并且具有不同的口音，所以决定给每个演讲者独一无二的身份。local/voxforge_fix_data.sh 根据提交日期将所有“匿名”演讲者重命名为“anonDDDD”（其中 D 代表十进制数字），这不完全精确。因为它可能会给录制在两个不同日期的同一位演讲者带来两个不同的 ID，并给予恰好在同一天发表演讲的两个或更多不同“匿名”演讲者相同的 ID。

将匿名演讲者映射到唯一 ID：

```
local/voxforge_map_anonymous.sh ${selected}
```

接下来将数据分成训练和测试集，并产生相关的转录和说话者依赖信息。这些步

骤由一个相当复杂而非特别有效的脚本（称为 local/voxforge_data_prep.sh）执行。 在 run.sh 中定义要分配给测试集的说话者数量，随机选择实际的说话者测试集。这可能不是一个理想的安排，因为每次启动 voxforge_data_prep.sh 时，说话者都会不同，并且测试集的 WER 也可能稍微不同。由于 VoxForge 仍然没有预定义的高质量训练和测试集及测试时语言模型，所以这不是很重要。

数据的初始归一化：

```
    local/voxforge_data_prep.sh --nspk_test ${nspk_test} ${selected}
```

使用 MITLM 工具包来评估测试时语言模型。用一个名为 local/voxforge_prepare_lm.sh 的脚本在 tools/mitlm-svn 中安装 MITLM，然后在训练集上训练出语言模型。

local/voxforge_prepare_dict.sh 是用于准备词典的脚本。它首先下载 CMU 的发音词典，并准备在训练集中有而不在 cmudict 中的单词列表。使用 Sequitur G2P 自动生成这些单词的发音，该工具安装在 tools/g2p 下。因为 Sequitur 模型的训练很花费时间，这个脚本会下载并使用一个在 cmudict 上预先构建好的模型。

解码脚本大部分是从 WSJ 和 RM 脚本中借用的。

1.8 数据准备

在顶层的 run.sh（例如，egs/rm/s5/run.sh）中有一些命令与数据准备各个阶段相关。名为 local/的子目录中的部分始终特定于数据库。例如，在资源管理（RM）设置中，它是 local/rm_data_prep.sh。在 RM 的情况下，这些命令为：

```
    local/rm data prep.sh /export/corpora5/LDC/LDC93S3A/rm comp || exit 1;
    utils/prepare lang.sh data/local/dict '!SIL' data/local/lang data/lang ||
exit 1;
    local/rm_prepare_grammar.sh || exit 1;
```

在 WSJ 脚本中，命令为：

```
    wsj0=/export/corpora5/LDC/LDC93S6B
```

```
    wsj1=/export/corpora5/LDC/LDC94S13B
    local/wsj_data_prep.sh $wsj0/??-{?,??}.? $wsj1/??-{?,??}.?  || exit 1;
    local/wsj prepare dict.sh || exit 1;
    utils/prepare lang.sh data/local/dict "<SPOKEN NOISE>" data/local/lang tmp
data/lang || exit 1;
    local/wsj_format_data.sh || exit 1;
```

在 WSJ 脚本中有更多的命令与本地的训练语言模型相关，但上面的命令是最重要的命令。

数据准备阶段的输出包括两个，一个涉及“数据”（目录如 data/train/），一个涉及“语言”（目录如 data/lang/）。“数据”部分与用户拥有的特定录音有关，而“语言”部分包含与语言本身相关的内容，例如词典、音素集合及有关 Kaldi 所需音素集合的各种额外信息。如果要准备使用现有系统和现有语言模型解码的数据，则只需接触“data”部分即可。

（1）“数据”部分的数据准备

训练数据位于 data/train 目录下；测试数据位于 data/eval2000 这样的目录下，和训练数据集具有基本相同的格式，只是可能在测试目录中有“.stm”和“.glm”文件，以启用 sclite 评分。sclite 是一款语音识别结果打分软件。

在 egs/swbd/s5 目录下查看一个电话总机的例子，执行命令为：

```
    s5# ls data/train
    cmvn.scp  feats.scp  reco2file and channel  segments  spk2utt  text  utt2spk
wav.scp
```

不是所有的文件都是同等重要的。简单的设置是没有分割信息的（即每个话语对应一个文件）。必须自行创建的文件包括 text、wav.scp 和 utt2spk，有时需要创建 segments 和 reco2file_and_channel，剩下的可由标准脚本创建。

text 文件包含每个话语的转录。在电话总机的例子中，这个文件为：

```
    s5# head -3 data/train/text
    sw02001-A 000098-001156 HI UM YEAH I'D LIKE TO TALK ABOUT HOW YOU DRESS FOR
WORK AND
    sw02001-A_001980-002131 UM-HUM
```

```
sw02001-A_002736-002893 AND IS
```

从上述第二行开始，每一行的第一个元素是话语 id，它是一个任意的文本字符串，但如果设置中有说话者信息，应该使 speaker-id 成为话语 id 的前缀，这对于与这些文件的排序有关的原因很重要。其余部分是每个话语的转录，不必确保所有单词都在词汇表中，词汇表中的单词将被映射到 data/lang/oov.txt 文件中指定的单词。

要注意 utt2spk 和 spk2utt 这两个文件的排序顺序一致性。例如，从 utt2spk（语音-id 所对应的说话者-id）文件提取的说话者-id 列表和字符串的排序顺序一样。最简单的做法就是说话者-id 作为语音-id 的前缀，一般用“-”分隔。如果说话者-id 长度不一致，在某些情况下用标准的 C 字符串排序时，说话者-id 和其对应的语音-id 可能以不同的顺序进行排序，这可能导致程序崩溃。

另外一个重要文件就是 wav.scp。在电话总机的例子中，这个文件为：

```
s5# head -3 data/train/wav.scp
sw02001-A /home/dpovey/kaldi-trunk/tools/sph2pipe v2.5/sph2pipe -f wav -p
-c 1 /export/corpora3/LDC/LDC97S62/swb1/sw02001.sph |
sw02001-B /home/dpovey/kaldi-trunk/tools/sph2pipe v2.5/sph2pipe -f wav -p
-c 2 /export/corpora3/LDC/LDC97S62/swb1/sw02001.sph |
```

格式为：

```
<recording-id> <extended-filename>
```

上述格式中 extended-filename 可能是确切的文件名，或者是提取.wav 格式文件的命令。extended-filename 最后的管道符号意味着它将被解读成管道。如果分割文件不存在，那么 wav.scp 每行的第一个字符就是语音-id。wav.scp 必须是单声道的，如果底层语音文件有多个声道，那么在 wav.scp 中就必须有一个短命令用来提取指定的通道。

在 Switchboard 设置中有 segments 文件，这个文件为：

```
s5# head -3 data/train/segments
sw02001-A_000098-001156 sw02001-A 0.98 11.56
sw02001-A_001980-002131 sw02001-A 19.8 21.31
sw02001-A_002736-002893 sw02001-A 27.36 28.93
```

格式为：

```
<utterance-id> <recording-id> <segment-begin> <segment-end>
```

上述格式中分别代表语音-id、记录-id、分割开始时间、分割结束时间，分割的开始和结束单位为 s。recording-id 是 wav.scp 中使用的相同标识，也是可以自行选择的任意标识。

reco2file_and_channel 文件只在打分（测量错误率）时用到。这个文件为：

```
s5# head -3 data/train/reco2file and channel
sw02001-A sw02001 A
sw02001-B sw02001 B
sw02005-A sw02005 A
```

格式为：

```
<recording-id> <filename> <recording-side (A or B)>
```

上述格式中 filename 为.sph 文件的名称，没有后缀，但一般来说，它是.stm 文件中的任何标识符。recording-side 代表一个电话对话中两个声道，如果不是，选择 A 更安全。如果没有.stm 文件或不知道干什么，就没必要创建 record2file_and_channel 文件。

date/train 目录下还需要自行创建的文件是 utt2spk 文件。这个文件为：

```
s5# head -3 data/train/utt2spk
sw02001-A_000098-001156 2001-A
sw02001-A_001980-002131 2001-A
sw02001-A_002736-002893 2001-A
```

格式为：

```
<utterance-id> <speaker-id>
```

说话者-id 不需要精确地对应每一个说话者，有个大概的猜测就好。可能会出现这样的情况——打电话的一方说了几句话后可能会将电话给另外一个人。如果没有说话者标识的相关信息，可以使说话者-id 和语音-id 相同，因此有人创建了 global 说话者-id（所有语音只对应一个说话者）。这样做不理想的原因在于：它使得 CMN 在训练时无效（因为被全局应用，一般 CMN 都是在句子内或者说一条语音内减去特征的均值），而且在使用 utils/split_data_dir.sh 分割数据时会引起问题。

在一些数据集中还存在其他的文件，例如与性别相关的文件为：

```
s5# head -3 ../../rm/s5/data/train/spk2gender
adg0 f
ahh0 m
ajp0 m
```

上述所有的文件都要排序，未排序会出现错误。排序的最终目的是使一些不支持 fseek()的数据流也能够达到随机访问的效果。许多 Kaldi 程序从其他 Kaldi 命令中读取很多管道、不同类型的对象，并且与合并排序的不同输入做比较。排序时要小心，将 shell 变量 LC_ALL 定义为“C”：

```
export LC_ALL=C
```

如果不这样做，文件的排序顺序将与 C++排序字符串的顺序不同，而 Kaldi 会崩溃。

如果数据中有.stm 和.glm 文件，使用 local/score.sh 脚本文件可以计算 WER。电话总机设置中会用到这些文件的打分脚本，例如 egs/swbd/s5/local/score_ sclite.sh 就会被 egs/swbd/s5/local/score.sh 调用。

可以从用户提供的文件生成相应目录中的其他文件。例如，utt2spk 转换成 spk2utt：

```
utils/utt2spk_to_spk2utt.pl data/train/utt2spk > data/train/spk2utt
```

因为 utt2spk 和 spk2utt 具有相同的信息，所以能够转换。

utt2spk 的格式：

```
<utterance-id><speaker-id>
```

spk2utt 的格式：

```
<spaker-id><utterance-id>…
```

接下来，查看 feats.scp 文件。这个文件为：

```
s5# head -3 data/train/feats.scp
sw02001-A_000098-001156 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_
train.1.ark:24
sw02001-A_001980-002131 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_
train.1.ark:54975
sw02001-A_002736-002893 /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_
```

```
train.1.ark:62762
```

该文件指向提取的特征。在电话总机例子中是指向 MFCC 特征，因为用了 make_mfcc.sh 脚本。

在 Kaldi 中每个特征文件都包含一个矩阵，在电话总机例子中矩阵的维度将会是 13 维（文件以 10ms 作为间隔）。/home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/raw_mfcc_train.1.ark:24 的意思是，打开.ark 文件，通过 fseek()文件找到第 24 个位置，从这开始读数据。

格式为：

```
<utterance-id> <extended-filename-of-features>
```

feats.scp 通过以下命令创建：

```
steps/make_mfcc.sh --nj 20 --cmd "$train_cmd" data/train exp/make_mfcc/train
$mfccdir
```

该脚本被顶层的 run.sh 脚本调用。shell 变量的定义可查看脚本说明。$mfccdir 是用户指定的目录，.ark 文件会写入到这个目录中。

data/train 目录的最后一个文件是 cmvn.scp。该文件包含倒谱均值和方差规整的数据，通过说话者标识进行索引。每组统计量都是一个矩阵，在电话总机例子中是维数 2×14 的矩阵。

cmvn.scp 文件中的部分内容如下：

```
s5# head -3 data/train/cmvn.scp
2001-A /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:7
2001-B /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:253
2005-A /home/dpovey/kaldi-trunk/egs/swbd/s5/mfcc/cmvn_train.ark:499
```

和 feats.scp 不同的是，该文件是通过说话者-id 进行索引的。该文件通过以下命令创建：

```
steps/compute_cmvn_stats.sh data/train exp/make_mfcc/train $mfccdir
```

为了防止数据准备阶段的错误引发后续问题，有工具可检查脚本是否符合格式规范：

```
utils/validate_data_dir.sh data/train
```

以下命令可以用来修复数据：

```
utils/fix_data_dir.sh data/train
```

此脚本将修复排序错误，并将删除缺少某些必需数据（如特征数据或讲稿）的任何话语。

（2）“语言”部分的数据准备

查看“lang”目录下的文件。

```
s5# ls data/lang
L.fst  L_disambig.fst  oov.int  oov.txt  phones  phones.txt  topo  words.txt
```

还有很多其他路径有相似的格式，例如在“data/lang_test”路径下就包含了相同的信息，还多了一个 G.fst 文件。

```
s5# ls data/lang test
G.fst  L.fst  L_disambig.fst oov.int  oov.txt phones phones.txt topo  words.txt
```

这些目录中的每一个似乎只包含几个文件，但实际不是那么简单，例如“phones”就是一个目录，而不是文件。

```
s5# ls data/lang/phones
context_indep.csl disambig.txt nonsilence.txt roots.txt silence.txt
context indep.int  extra questions.int  optional silence.csl  sets.int  word
boundary.int
context_indep.txt  extra_questions.txt  optional_silence.int  sets.txt  word_
boundary.txt
disambig.csl nonsilence.csl optional_silence.txt silence.csl
```

phones 目录包含音素集合的各种信息，其中一些文件有 3 个不同的版本，扩展名为.csl、.int 和.txt，这 3 种格式包含了相同的信息，不需要全部创建。因为通过一个简单的输入，utils/prepare_lang.sh 脚本就会输出这些文件。

接下来创建 lang 目录。data/lang/目录包含很多不同的文件，所以提供了脚本从一些简单的输入开始来创建这些文件。

```
utils/prepare_lang.sh data/local/dict "<UNK>" data/local/lang data/lang
```

这里的输入是目录 data/local/dict/，标签 <UNK>用来映射出现在讲稿中的 OOV 单

词（这会成为 data/lang/oov.txt）。data/local/lang/是脚本会用到的临时目录；data/lang/是脚本的输出目录。

数据准备者需要创建的是目录 data/local/dict/。此目录包含以下内容：

```
s5# ls data/local/dict
extra questions.txt  lexicon.txt nonsilence phones.txt  optional silence.txt
silence_phones.txt
```

实际上还有几个没有列出的文件，但它们只是创建 data/lang 目录放进去的临时文件，可以忽略它们。extra_questions.txt 是一个空文件。nonsilence_phones.txt 和 silence_phones.txt 分别用于分开“真实”音素和“静音”音素。

nonsilence_phones.txt 文件中的内容如下：

```
s5# head -3 data/local/dict/nonsilence phones.txt
IY
B
D
```

silence_phones.txt 文件中的内容如下：

```
s5# cat data/local/dict/silence_phones.txt
SIL
SPN
NSN
LAU
```

lexicon.txt 文件中的内容如下：

```
s5# head -5 data/local/dict/lexicon.txt
!SIL SIL
-'S S
-'S Z
-'T K UH D EN T
-1K W AH N K EY
```

格式为：

```
<word> <phone1> <phone2> ...
```

注意：如果同一个词有不同发音，则 lexicon.txt 中可能会包含分布在不同行的多

个重复条目。如果想使用发音概率，创建 lexiconp.txt（第二个字段是概率）代替原来的 lexicon.txt。注意一般都要对发音概率进行规范化，这样可以让每个单词最有可能的发音总是一个，从而得到更好的结果。对于以发音概率运行的顶层脚本，可以在 egs/wsj/s5/run.sh 中通过关键字“pp”找到。

文件 extra_questions.txt 涉及重音标记或音调标记的一些内容，用户可能会需要具有不同重音或音调的特定音素的不同版本。为了演示这样的内容，查看与上述相同的文件，但在 egs/wsj/s5/中。结果如下：

```
s5# cat data/local/dict/silence phones.txt
SIL
SPN
NSN
s5# head data/local/dict/nonsilence phones.txt
S
UW UW0 UW1 UW2
T
N
K
Y
Z
AO AO0 AO1 AO2
AY AY0 AY1 AY2
SH
s5# head -6 data/local/dict/lexicon.txt
!SIL SIL
<SPOKEN NOISE> SPN
<UNK> SPN
<NOISE> NSN
!EXCLAMATION-POINT  EH2 K S K L AH0 M EY1 SH AH0 N P OY2 N T
"CLOSE-QUOTE  K L OW1 Z K W OW1 T
s5# cat data/local/dict/extra_questions.txt
SIL SPN NSN
S UW T N K Y Z AO AY SH W NG EY B CH OY JH D ZH G UH F V ER AA IH M DH L AH
P OW AW HH AE R TH IY EH
UW1 AO1 AY1 EY1 OY1 UH1 ER1 AA1 IH1 AH1 OW1 AW1 AE1 IY1 EH1
```

```
    UW0 AO0 AY0 EY0 OY0 UH0 ER0 AA0 IH0 AH0 OW0 AW0 AE0 IY0 EH0
    UW2 AO2 AY2 EY2 OY2 UH2 ER2 AA2 IH2 AH2 OW2 AW2 AE2 IY2 EH2
```

可以看到，nonsilence_phones.txt 中有些单行中包括多个音素。这些是元音的不同重音相关版本。注意，CMU 字典中每个音素都出现了 4 个不同版本，例如，UW UW0 UW1 UW2，由于某些原因，其中一个版本没有数字后缀。音素在一行的顺序并不重要，但分组到不同行时就很重要了；一般来说，Kaldi 建议“真实音素”分组的不同形式在不同行。Kaldi 使用 CMU 字典中存在的重音标记。文件 extra_questions.txt 包括一个含所有“silence”音素的问题（实际上这是不必要的，因为脚本 prepare_lang.sh 会添加这样的问题），还有一个与每个不同的重音标记相对应的问题。为了从重音标记得到更好的结果，这些问题是必要的，因为实际上每个音素的不同重音相关版本都统一放在 nonsilence_phones.txt 文件行中，确保它们都在文件 data/lang/phones/roots.txt 和 data/lang/phones/sets.txt 中，这反过来又确保它们共享相同的树根结点，并且永远不能被一个问题所区分。因此，必须提供一个特殊的问题，使决策树建立过程能够区分音素。注意，将音素放在 sets.txt 和 roots.txt 中的原因是，一些重音相关版本的音素可能没有足够的数据可靠地估计单独的决策树或生成问题时用到的音素聚类信息。通过将它们分组在一起，确保在没有足够数据的情况下分别预测它们，这些不同版本的音素在整个决策树构建过程中都“保持在一起”。

还有一点要提出的是脚本 utils/prepare_lang.sh 支持几个选项。以下是脚本的使用信息，可以看看这些选项到底是什么。

```
    usage: utils/prepare lang.sh <dict-src-dir> <oov-dict-entry> <tmp-dir> <
lang-dir>
    e.g.: utils/prepare lang.sh data/local/dict <SPOKEN NOISE> data/local/la
ng data/lang
   options:
     --num-sil-states <number of states>      #默认: 5, #静音模型中的状态
     --num-nonsil-states <number of states>   #默认: 3, #非静音模型中的状态
     --position-dependent-phones (true|false) #默认: true; 如果是 true,则使用
 B,  E,  S & I
                                              #音素上的标记表示单词内部位置
```

```
    --share-silence-phones (true|false)      #默认：false；如果为 true，则共享所有非静音音素的 pdfs
    --sil-prob <probability of silence>      #默认：0.5 [必须保证 0 < silprob < 1]
```

一个潜在的重要选项是-share-silence-phones 选项，其默认值为 false。如果该选项为 true，则将共享所有静音音素的所有 PDF（高斯混合模型），例如静音、发声噪声、噪声和笑声，并且这些模型之间仅转换概率不同。它对 IARPA BABEL 项目的粤语数据非常有帮助。这些数据非常混乱且有很长的未转录部分，所以试图将其与为此目的指定的特殊音素对齐。训练数据在某种程度上可能无法正确对齐，并且由于某种原因，将此选项设置为 true 会改变未对齐的情况。

创建 G.fst 文件。实际上，在某些设置中，可能有许多“lang”目录用于测试目的，具有不同的语言模型和词典。华尔街日报（WSJ）的设置就是一个例子。

```
    s5# echo data/lang*
    data/lang data/lang test bd fg data/lang test bd tg data/lang test bd tgpr data/lang test bg \
    data/lang test bg 5k data/lang test tg data/lang test tg 5k data/lang test tgpr data/lang_test_tgpr_5k
```

根据是使用统计语言模型还是使用某种语法，创建 G.fst 文件的过程是不同的。在 RM 设置中有一个 bigram 语法，它只允许某些单词对。通过在输出边的数量上分配 1 的概率，使这个和在每个文法状态中为 1。local/rm_data_prep.sh 中有一个语句，它执行以下操作：

```
    local/make_rm_lm.pl  $RMROOT/rm1_audio1/rm1/doc/wp_gram.txt   > $tmpdir/G.txt || exit 1;
```

此脚本 local/make_rm_lm.pl 以 FST 格式创建语法（是文本格式，而不是二进制格式），它包含如下行：

```
    s5# head data/local/tmp/G.txt
    0    1    ADD    ADD    5.19849703126583
    0    2    AJAX+S    AJAX+S    5.19849703126583
    0    3    APALACHICOLA+S    APALACHICOLA+S    5.19849703126583
```

脚本 utils/format_lm.sh 把 ARPA 格式的语言模型转换成 OpenFST 格式类型。该脚

本用法如下：

```
    Usage: utils/format lm.sh <lang dir> <arpa-LM> <lexicon> <out dir>
    E.g.:  utils/format lm.sh  data/lang  data/local/lm/foo.kn.gz  data/local/
dict/lexicon.txt data/lang test
    Convert ARPA-format language models to FSTs.
```

该脚本的一些关键命令如下：

```
    gunzip -c $lm \
      | arpa2fst --disambig-symbol=#0 \
               --read-symbol-table=$out_dir/words.txt - $out_dir/G.fst
```

Kaldi 程序 arpa2fst 将 ARPA 格式的语言模型转换成一个加权有限状态转换器（实际上是接收器）。

1.9　加权有限状态转换

因为大多数组件（语言模型、词典和词图）都是可以用有限状态机描述，可以通过组合操作将不同的模型集成到一个模型中，所以采用加权有限状态转换（WFST）。WFST 是用于描述模型的统一框架。

级联多个加权有限状态转换如下。

H：HMM

C：上下文相关模型

L：词典

G：文法

级联 4 个 WFST 的形式化写法为：

```
    H ○ C ○ L ○ G
```

只需要将这个识别网络（WFST 网络）读入内存，然后基于声学模型就可以在这个网络上完成解码，不需要像原有系统那样同时考虑声学模型、词典和语言模型等。这样简化了语音识别系统的设计与实现。

1.9.1　FSA

有限状态接收器（Finite State Acceptor，FSA）接收一个字符串的集合。一个字符串是一个符号序列。对于给定的字符串，FSA 返回“接收”或“不接收”两种结果。将 FSA 可视为无限个字符串集的表示，如图 1-2 所示。

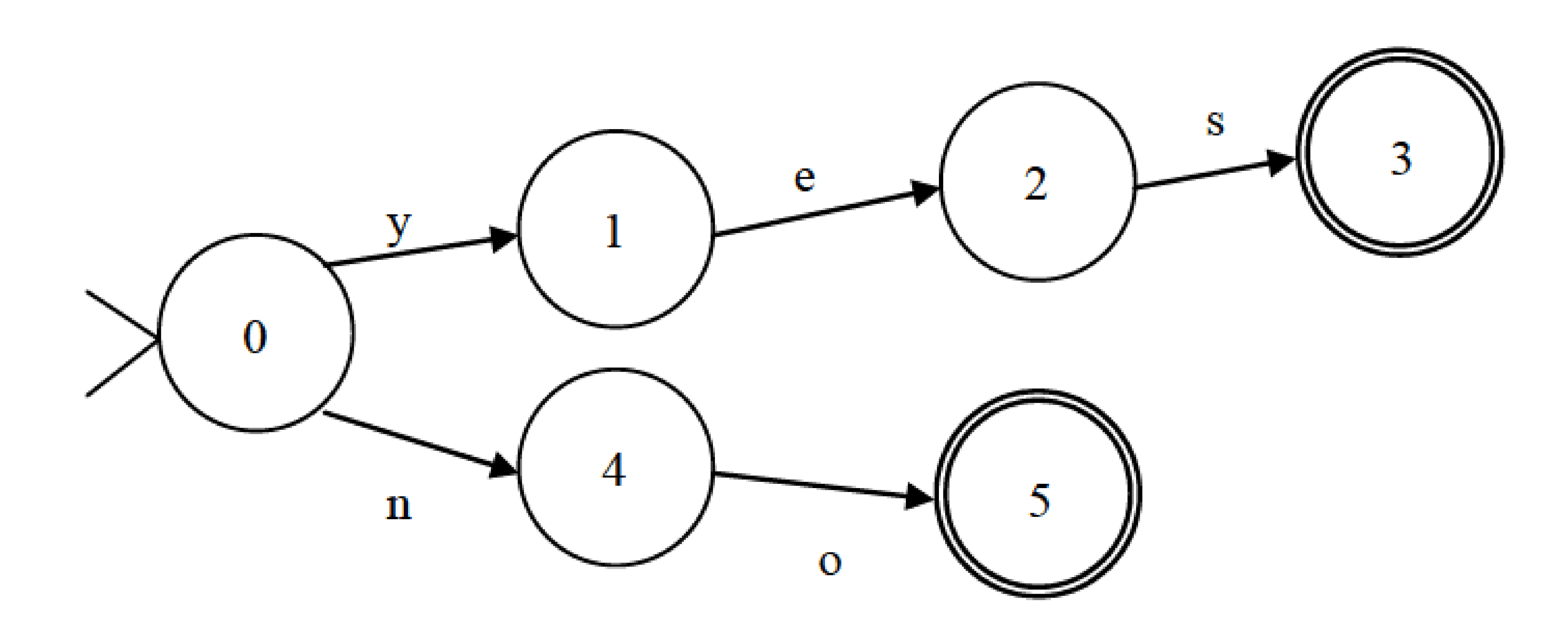


图 1-2　表示 yesno 的有限状态接收器

图 1-2 中的 FSA 只接受字符串“yes”和“no”，即 set{yes,no}；圈子中的数字是状态标签（不是很重要）；标签是在边上的符号；箭头指向的结点是开始结点，双圈结点表示可以作为结束结点。开始结点只能有一个，而结束结点可以有多个。这样的图称为状态转换图。

dk.brics.automaton 是一个 FSA 的实现。使用它定义表示 yesno 的 FSA 代码如下：

```
String s1 = "yes";
String s2 = "no";
Automaton a= Automaton.makeString(s1);        //yes 有限状态接收器
Automaton b= Automaton.makeString(s2);        //no 有限状态接收器
Automaton c = BasicOperations.union(a, b);    //并运算
String word = "yes";
System.out.println(c.run(word));
```

1.9.2　FST

有限状态转换器（FST）就是利用有限状态机把输入串映射成输出串。

例如，判断二进制串的奇偶性，状态转换示意图如图 1-3 所示。其中用两个状态

S1 和 S2 分别表示偶数和奇数。

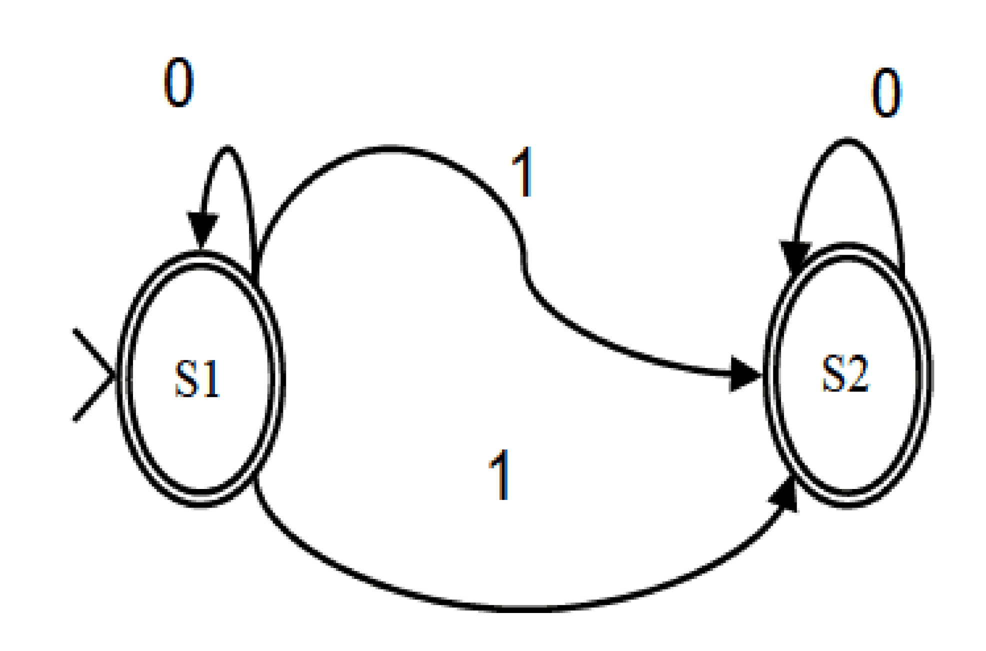


图 1-3　状态转换示意图

再如，要求以下字符串中“1”的数量是奇数还是偶数。

1 0 1 1 0 0 1 → S1

0 0 0 1 0 0 0 → S2

状态转换表如表 1-1 所示。

表 1-1　状态转换表

状　　态	转　　换
S1	0 → S1, 1 → S2
S2	0 → S2, 1 → S1

实现该有限状态转换器的代码如下：

```
int parity(String s) {
   int state = 1;
   for(int i = 0;i< s.length();++i){
      char ch = s.charAt(i);
      switch (state){
        case 1:
           if(ch=='1')
             state = 2;
           break;
        case 2:
           if(ch=='1')
             state = 1;
           break;
```

```
            }
        }
        return state;
    }
```

测试这个方法，代码为：

```
System.out.print(parity("01010")); //输出 1
```

为了构建最小完美散列，需要把排好序的单词{clear,clever,ear,ever,fat,father}（见图 1-4）映射到序号（0, 1, 2,⋯）。当遍历边时，把经过的值加起来，例如“father”在“f”命中 4 且在“h”命中 1 时，输出 5。

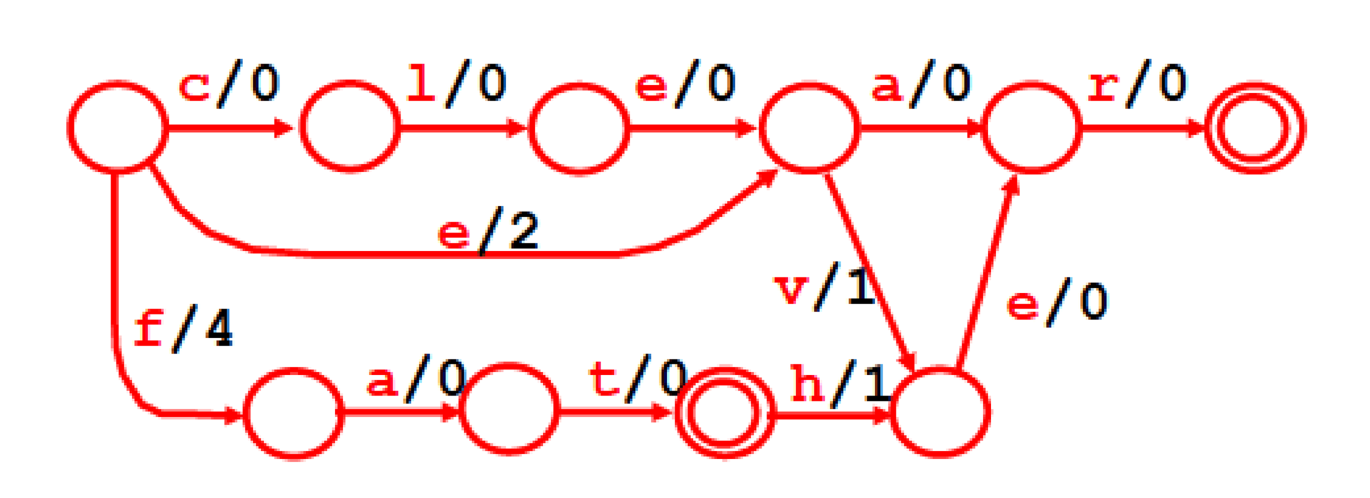


图 1-4 有限状态转换器

1.9.3 WFST

环是抽象代数中使用的基本代数结构之一，它由一个装备有两个二元运算的集合组成。半环是与环相似的代数结构，但不要求每个元素必须具有加法可逆。

- 求和：计算序列的权重（用该序列标记的路径权重之和）。
- 乘积：计算路径的权重（构成转换权重的乘积）。

常见的半环比较如表 1-2 所示。

表 1-2 常见的半环比较

名称	范围	⊕（求和）	⊗（乘积）	$\bar{0}$	$\bar{1}$
Boolean	{0,1}	∨	∧	0	1
Real	[0,∞]	+	*	0	1
Log	[−∞,∞]	−log (exp(−a) + exp(−b))	+	∞	0
Tropical	[−∞,∞]	min	+	∞	0

所有实际应用都使用热带半环（Tropical Semiring），其最明显的实例是在语音识别系统中假设组合词语的负对数概率。

以下是在 jopenfst 中使用热带半环的例子。

```
MutableFst fst = new MutableFst(TropicalSemiring.INSTANCE);
//用符号识别状态
fst.useStateSymbols();
MutableState startState = fst.newStartState("<start>");
//设置最终权重让这个状态成为合格的最终状态
fst.newState("</s>").setFinalWeight(0.0);
//可以将符号手动添加到符号表
int symbolId = fst.getInputSymbols().getOrAdd("<eps>");
fst.getOutputSymbols().getOrAdd("<eps>");
//直接加边
fst.addArc("state1", "inA", "outA", "state2", 1.0);
//也可以使用状态实例
fst.addArc(startState, "inC", "outD", fst.getOrNewState("state3"), 123.0);
```

1.9.4 Kaldi 对 OpenFst 的改进

Kaldi 将 OpenFst 代码本身用于许多算法。Kaldi 对 OpenFst 的改进算法位于目录 src/fstext 中，相应的命令行程序存在于 fstbin/中。此代码使用了 OpenFst 库，仅在这里描述当前实际使用的算法。

- Kaldi 使用与 OpenFst 中的算法不同的确定化算法，需要将其名称与函数 DeterminizeStar()区分开来，并将相应的命令行程序命名为 fstdeterminizestar。Kaldi 的确定化算法实际上比 OpenFst 中的确定化算法更接近标准 FST 确定化算法，因为它在确定化的过程中也去除了 epsilon（与许多其他 FST 算法一样，Kaldi 不认为 epsilon 是“真实符号”）。
- Kaldi 提供了一个在 log 半环中的确定化函数 DeterminizeInLog()，它在确定化之前，将正常（热带）半环中的 FST 投射到 log 半环，然后转换回来。

- Kaldi 还提供一个名为 RemoveEpsLocal()的 epsilon 去除算法，保证永远不会毁掉 FST，但另一方面不保证删除所有 epsilons。从本质上讲，它可以移除任何可以轻松移除的 epsilons 而不会使图形变大。函数 RemoveEpsLocal()保留 FST 等价性。
- Kaldi 使用 OpenFst 提供的最小化算法，但是在编译 OpenFst 之前应用补丁，以便最小化可应用于非确定性 FST。

在大多数情况下，Kaldi 使用 OpenFst 自己的组合算法，但 Kaldi 使用函数 TableCompose()和相应的命令行程序 fsttablecompose 是一种针对某些常见情况的更有效的组合算法。当与具有非常高出度的词典组合时，使用 TableCompose()可以提高组合速度。

1.10 语音识别语料库

本节讲解几个常用的语音识别语料库，更多的语料库可以从 OpenSLR 网站（http://www.openslr. org/）下载。OpenSLR 是一个致力于托管语音和语言资源的站点，例如用于语音识别的训练语料库和与语音识别相关的软件。

1.10.1 TIMIT 语料库

TIMIT 语料库有着准确的音素标注，可以应用于语音分割性能评价，同时含有几百个说话者语音，所以它也是评价说话者语音识别常用的权威语料库。

TIMIT 语料库旨在提供语音数据和自动语音识别系统的开发和评估。TIMIT 包含 630 个说话者的宽带录音，8 个主要方言区的美式英语，每个人阅读 10 个语音丰富的句子。TIMIT 语料库包括时间对齐的单词内容、语音和单词转录及每个话语的 16 位、16kHz 语音波形文件。语料库设计是麻省理工学院（MIT）、斯坦福国际研究院（SRI）和得州仪器公司（TI）共同的努力成果。演讲在得州仪器公司录制，转录在麻省理工

学院，并由美国国家标准技术研究所（NIST）验证。

1.10.2 LibriSpeech 语料库

LibriSpeech 语料库是一个大型英语阅读语料库。来自 LibriVox 项目的有声读物，采样频率为 16kHz。该库的口音是多种多样的，没有标记，但大多数是美式英语。LibriSpeech 语料库还有单独准备好的语言模型训练数据和预建好的语言模型。

1.10.3 中文语料库

中文的语音识别公共数据集共有以下 3 个。

- gale_mandarin：中文新闻广播数据集。
- hkust：中文电话数据集。
- thchs30：清华大学 30 小时数据集。

有些数据集中包含 Linux 链接文件。

1.11 Linux shell 脚本基础

shell 是用户和 Linux 内核之间的接口程序，用户在命令行提示符下输入的每个命令都由 shell 先解释后传给 Linux 内核。shell 是一款命令语言解释器，拥有内置的 shell 命令集。此外，shell 也能被系统中其他有效的 Linux 实用程序和应用程序所调用。

shell 的主要功能包括以下几个。

- 命令解释功能：将用户可读的命令转换成计算机可理解的命令，并控制命令执行。
- 输入/输出重定向：操作系统将键盘作为标准输入、显示器作为标准输出，当这些定向不能满足用户需求时，用户可以在命令中用符号“>”或“<”重新定向。
- 管道处理：利用管道将一条命令的输出送入另一条命令，实现多条命令组合完

成复杂功能。

- 系统环境设置：用 shell 命令设置环境变量，维护用户的工作环境。
- 程序设计语言：shell 命令本身可以作为程序设计语言，将多个 shell 命令组合起来，编写能实现系统或用户所需功能的程序。

市面上有多种 shell，例如 zshell 和 fish 等，一般使用 Bash 脚本。

1.11.1 Bash

在屏幕上打印“Hello”：

```
echo "Hello"
```

将 ABC 分配给 a：

```
a=ABC
```

输出 a 的值：

```
echo $a
```

此时，在屏幕上打印 ABC。

将 ABC.log 分配给 b：

```
b=$a.log
```

输出 b 的值：

```
#echo $b
ABC.log
```

把文件“ABC.log”的内容写入到 testfile：

```
cat $b > testfile
```

指令“--help”会输出帮助信息。

可以把重复执行的 shell 脚本写入到一个文本文件。在 Linux 中，文件后缀名不作为系统识别文件类型的依据，但是可作为用户识别文件的依据，可以简单地将脚本文件以.sh 结尾。在 Linux 下，可以通过 vi 命令创建一个诸如 script.sh 的文件，即 vi script.sh。

创建脚本文件后就可以在文件内用脚本语言要求的格式编写脚本程序了。

在创建的脚本文件中输入以下代码并保存和退出。

```
#! /bin/bash
echo "hello world!"
```

添加脚本文件的可执行运行权限 chmod 777 script.sh 后，运行文件./script.sh 得到以下结果：

```
hello world!
```

注意：shell 脚本中用“#”表示注释符，相当于 C 语言中的注释符“//”。但如果“#”位于第一行开头，并且是“#!”（称为 Shebang）形式则例外，它表示该脚本使用后面指定的解释器/bin/sh 解释执行。每个脚本程序必须在开头包含这个语句。

使用参数 n 检查语法错误，例如：

```
#bash -n ./test.sh
```

如果 shell 脚本有语法错误，则会提示错误所在行；否则，不输出任何信息。

if 语句的语法格式：

```
if [ condition ] then
      command1
elif   #和 else if 等价
   then
      command2
   else
      default-command
fi
```

这里的 fi 就是 if 反过来写。

为了判断某个命令是否存在，可以使用如下的代码：

```
if which programname >/dev/null; then
    echo exists
else
    echo does not exist
fi
```

判断 yum 是否存在，可以使用如下的代码：

```
if which yum >/dev/null; then
    echo "exists"
else
    echo "does not exist"
fi
```

case 语句的语法格式：

```
case 字符串 in
    模式 1)
        语句
        ;;
    模式 2)
         语句
         ;;
    *)
         默认执行的语句
         ;;
esac
```

这里的 esac 就是 case 反过来写。

例如：

```
extension="png"
case "$extension" in
  "jpg"|"jpeg")
    echo "It's image with jpeg extension."
  ;;
  "png")
    echo "It's image with png extension."
  ;;
  "gif")
    echo "Oh, it's a giphy!"
  ;;
  *)
    echo "Woops! It's not image!"
  ;;
esac
```

这里使用“|”把"jpg"和"jpeg"这两个格式连接到了一起。

下面介绍 4 种模式匹配。

- ${variable#pattern}：从$string 的前面删除$substring 的最短匹配。
- ${variable##pattern}：从$string 的前面删除$substring 的最长匹配。
- ${variable%pattern}：从$string 的后面删除$substring 的最短匹配。
- ${variable%%pattern}：从$string 的后面删除$substring 的最长匹配。

使用模式匹配的例子：

```
x=/home/cam/book/long.file.name
echo ${x#/*/}
echo ${x##/*/}
echo ${x%.*}
echo ${x%%.*}
   cam/book/long.file.name
   long.file.name
   /home/cam/book/long.file
   /home/cam/book/long
```

1.11.2 AWK

典型的 AWK 程序充当过滤器，从标准输入读取数据，并输出标准的过滤数据。它一次读取数据的一条记录。默认情况下，一次读取一行文本。每次读取记录时，AWK 自动将记录分隔到字段中。字段在默认情况下也是由空白分隔的。每个字段被分配给一个变量，该变量有一个数字名称。变量 $0 表示整个记录，$1 表示第一个字段，$2 表示第二个字段，依此类推。此外，还设置了一个名为 NF 的变量，其中包含在记录中检测到的字段数量。下面来试试一个很简单的例子。

过滤 ls 命令的输出，具体代码如下：

```
#ls -l ./ | awk '{print $0}'
```

显示文本文件 nohup.out 匹配（含有）字符串"sun"的所有行，具体代码如下：

```
#awk '/sun/{print}' nohup.out
```

由于显示整个记录（全行）是 awk 的缺省操作，因此可以省略 action 项。

再如，要得到 Python 的版本号，可以使用如下的代码：

```
#python 2>&1 --version | awk '{print $2}'
2.7.5
```

这里的“2>&1”的含义是，把标准错误重定向到标准输出。

第 2 章 C#开发语音识别

记录语音、准备训练集是开发语音识别系统的基础。本章先讲解如何使用 NAudio 记录语音等相关内容。

2.1 准备开发环境

C#代码可以运行在.Net Framework，也可以运行在 mono 环境中。使用如下命令安装 mono 的安装包 mono-5.4.1.6-x64-0.msi 到 d:\mono\目录下。

```
msiexec /i "mono-5.4.1.6-x64-0.msi" INSTALLFOLDER="d:\mono\" /qb
```

C#开发环境可以使用 Visual Studio .Net，这里使用 Visual Studio 2017 社区版。Visual Studio 2017 可以到微软公司的网站 https://www.visualstudio.com/downloads/下载得到。新建一个控制台类型的解决方案会自动生成一个源代码文件。C#源文件的扩展名为.cs，如语音类 Audio.cs，或者语音文件工具类 WavFileUtils.cs。标识符的名称不能随便改，因为可能有很多地方用到同一个标识符。Visual Studio 支持通过重构重命名标识符。例如，要修改一个变量的名称，只需要将光标放在变量的名称上面，然后从“重构”菜单中选择“重命名”命令，在弹出的对话框中输入变量的新名称。

2.2　计算卷积

语音信号可以使用一维卷积来处理。把输入向量用 f 表示，卷积核用 g，并且假设 f 的长度为 n，g 的长度为 m，则 f 和 g 的卷积 $f*g$ 定义为：

$$(f*g)(i)=\sum_{j=i}^{m}g(j)\times f\left(i-j+\frac{m+1}{2}\right)$$

特殊地，如果卷积核的长度为 1，而且值也为 1，则 $f*g=f$。

一维卷积实现代码如下：

```
public static int[] convolute(int[] x,              //输入信号
                               int[] h)             //卷积核
{
    int[] y = new int[x.Length + h.Length - 1];    //输出卷积结果
    int[] negH = h.Reverse().ToArray();            //反转
    for (int i = 0; i < x.Length + h.Length - 1; i++)
    {
        int k = i;
        for (int j = h.Length - 1; j >= 0; j--)
        {
            if (k >= 0 && k < x.Length)
            {
                y[i] += x[k] * negH[j];
            }
            k--;
        }
    }
    return y;  //返回卷积结果
}
```

测试这个方法：

```
int[] x = { 1, 0, 2, 3, 0, 1, 1 };    //输入
int[] h = { 2, 1, 3 };                //卷积核
int[] output = convolute(x, h);
```

```
    Console.WriteLine(string.Join(", ", output));
```

输出卷积结果：

```
    2, 1, 7, 8, 9, 11, 3, 4, 3
```

2.3 记录语音

使用 NAudio 可以捕获进入计算机的话筒（或线路输入）的声音，并将捕获的声音录制到 WAV 文件中。用 WaveIn 可以在 WinForms 应用程序中录制 WAV 文件。在下面这个例子中，将看到如何创建一个非常简单的 WinForms 应用程序，将音频记录到 WAV 文件。

选择录制音频的位置。音频将被记录到桌面上 NAudio 文件夹中名为 recorded.wav 的文件中。

```
    var outputFolder =
    Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.Desktop),
"NAudio");
    Directory.CreateDirectory(outputFolder);
    var outputFilePath = Path.Combine(outputFolder,"recorded.wav");
```

创建录制设备。在这里，将使用 WaveInEvent，也可以使用 WaveIn 或 WasapiCapture。

```
    var waveIn = new WaveInEvent();
```

声明 WaveFileWriter，但直到开始录制时，才会创建它。

```
    WaveFileWriter writer = null;
```

设置窗口。它有两个按钮——开始按钮和停止录制按钮。声明一个关闭标志，以便在窗口关闭时停止录制。

```
    bool closing = false;                                   //关闭标志
    var f = new Form();                                     //窗口
    var buttonRecord = new Button() { Text = "Record" };    //开始按钮
    var buttonStop = new Button() { Text = "Stop", Left = buttonRecord.Right,
Enabled = false };                                          //停止录制按钮
```

```
f.Controls.AddRange(new Control[] { buttonRecord, buttonStop });
```

我们需要一些事件处理器。当单击开始按钮时，将创建一个新的 WaveFileWriter，指定要创建的 WAV 文件的路径及录制的格式。录制格式必须与录制设备格式相同。因此，我们使用 waveIn.WaveFormat。然后，使用 waveIn.StartRecording()开始进行录制，并适当地设置按钮启用状态。

```
buttonRecord.Click += (s, a) =>
{
    writer = new WaveFileWriter(outputFilePath, waveIn.WaveFormat);
    waveIn.StartRecording();
    buttonRecord.Enabled = false;
    buttonStop.Enabled = true;
};
```

还需要处理器来处理输入设备上的 DataAvailable 事件。开始录制后，会定期触发该事件。可以将事件参数中的缓冲区写入 writer，并确保写入了 a.BytesRecorded 字节，而不是 a.Buffer.Length。

```
waveIn.DataAvailable += (s, a) =>
{
    writer.Write(a.Buffer, 0, a.BytesRecorded);
};
```

录制 WAV 文件时经常添加的一项安全功能是限制 WAV 文件的大小。这个文件的容量迅速变大，但是无论如何，都不能超过 4GB。在这里，设置 30s 后会停止录制。

```
waveIn.DataAvailable += (s, a) =>
{
    writer.Write(a.Buffer, 0, a.BytesRecorded);
    if (writer.Position > waveIn.WaveFormat.AverageBytesPerSecond * 30)
    {
        waveIn.StopRecording();
    }
};
```

处理停止录制按钮。这很简单，只需要调用 waveIn.StopRecording()。但是，仍然可以

在 DataAvailable 回调中接收更多数据，因此请不要处理 WaveFileWriter。

```
buttonStop.Click += (s, a) => waveIn.StopRecording();
```

还将添加一项安全措施，如果在录制过程中尝试关闭窗口，则会调用 StopRecording() 并设置一个标记，以便知道，也可以处理输入设备。

```
f.FormClosing += (s, a) => { closing=true; waveIn.StopRecording(); };
```

为了安全地处理 WaveFileWriter（需要做的是生成一个有效的 WAV 文件），应该在录制设备上处理 RecordingStopped 事件。处理 WaveFileWriter，以便它修复 WAV 文件中的头信息，以使其有效。然后，设置按钮状态。最后，如果正在关闭窗口，则应该处理输入设备。

```
waveIn.RecordingStopped += (s, a) =>
{
    writer?.Dispose();
    writer = null;
    buttonRecord.Enabled = true;
    buttonStop.Enabled = false;
    if (closing)
    {
        waveIn.Dispose();
    }
};
```

所有的处理器都已设置好了，下面我们准备显示对话框。

```
f.ShowDialog();
```

完整的程序代码如下：

```
var outputFolder =
    Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.Desktop),
"NAudio");
Directory.CreateDirectory(outputFolder);
var outputFilePath = Path.Combine(outputFolder,"recorded.wav");
var waveIn = new WaveInEvent();
WaveFileWriter writer = null;
```

```
bool closing = false;
var f = new Form();
var buttonRecord = new Button() { Text = "Record" };
var buttonStop = new Button() { Text = "Stop", Left = buttonRecord.Right,
    Enabled = false };
f.Controls.AddRange(new Control[] { buttonRecord, buttonStop });
buttonRecord.Click += (s, a) =>
{
    writer = new WaveFileWriter(outputFilePath, waveIn.WaveFormat);
    waveIn.StartRecording();
    buttonRecord.Enabled = false;
    buttonStop.Enabled = true;
};
buttonStop.Click += (s, a) => waveIn.StopRecording();
waveIn.DataAvailable += (s, a) =>
{
    writer.Write(a.Buffer, 0, a.BytesRecorded);
    if (writer.Position > waveIn.WaveFormat.AverageBytesPerSecond * 30)
    {
        waveIn.StopRecording();
    }
};
waveIn.RecordingStopped += (s, a) =>
{
    writer?.Dispose();
    writer = null;
    buttonRecord.Enabled = true;
    buttonStop.Enabled = false;
    if (closing)
    {
        waveIn.Dispose();
    }
};
f.FormClosing += (s, a) => { closing=true; waveIn.StopRecording(); };
f.ShowDialog();
```

2.4 读入语音信号

声音经过模拟设备记录或再生成为模拟音频，再经数字化成为数字音频。PCM（Pulse Code Modulation，脉冲编码调制）文件是模拟音频信号经模-数转换直接形成的二进制序列。

PCM 流具有两个基本属性来确定流对原始模拟信号的保真度——采样率，即每秒取样的次数及确定可用于表示每个样本可能数字值数量的比特深度。大多数存储的未压缩音频是 16 位。其他位深度，如 8 和 24 也是常见的，并且存在许多其他位深度。

数字化时的采样率必须高于信号带宽的两倍，才能正确恢复信号。1Hz 代表每秒钟采样 1 次。声音采样频率一般为 8kHz，也就是每秒采样 8000 次。人们能够听见的音频频率范围为 60Hz～20kHz，其中语音分布在 300Hz～4kHz 内，而音乐和其他自然声音是全范围分布的。识别语音的最小频率范围为 300Hz～4kHz。

由于 16 位深度很常见，所以以此为例来了解数据是如何格式化的。通常将 16 位音频存储为打包的 16 位有符号整数。整数可能是 big-endian（最常见的是 AIFF）或 little-endian（最常见的是 WAV）。如果有多个通道，通道间通常是交错的。例如，在立体声音频中，有一个表示左声道的 16 位整数，后面跟着一个代表右声道的 16 位整数。这两个样本代表同一时间，两者一起有时称为采样帧或简单称为帧。short 数据类型表示 16 位有符号整数。因此，要读取原始 16 位数据，通常需要将数据定义为一个 short 类型的数组。

例如，有一个 WAV 文件（16 位 PCM：44kHz 两通道），现为两个通道中的每一个提取采样到两个 short 类型的数组。

```
using (WaveFileReader pcm = new WaveFileReader(@"file.wav"))
{
    int samplesDesired = 5000;
    byte[] buffer = new byte[samplesDesired * 4];
    short[] left = new short[samplesDesired];  //左声道数组
    short[] right = new short[samplesDesired]; //右声道数组
    int bytesRead = pcm.Read(buffer, 0, 10000);
```

```
        int index = 0;
        for(int sample = 0; sample < bytesRead/4; sample++)
        {
            left[sample] = BitConverter.ToInt16(buffer, index);
            index += 2;
            right[sample] = BitConverter.ToInt16(buffer, index);
            index += 2;
        }
    }
```

为了方便后续处理，可以将数据归一化成为-1～1 的浮点数。一般的语音文件为 16 位深度，也就是-1～1 对应-32 768～32 767 的整数。

2.5 离散傅里叶变换

本节计算给定复数向量的离散傅里叶变换（DFT）。因为要用到 System.Numerics.dll 中的 Complex 结构，所以在项目中增加对 System.Numerics 的引用。实现代码如下：

```
//输入是.NET 内部的复数(Complex)数组
public static Complex[] computeDft(Complex[] input) {
   int n = input.Length;
   Complex[] output = new Complex[n];
   for (int k = 0; k < n; k++) {      //对于每个输出元素
      Complex sum = 0;
      for (int t = 0; t < n; t++) {   //对于每个输入元素
         double angle = 2 * Math.PI * t * k / n;
         sum += input[t] * Complex.Exp(new Complex(0, -angle));
      }
      output[k] = sum;
   }
   return output;
}
```

输入是普通数值的实现代码如下：

```
public static void computeDft(double[] inreal, double[] inimag,
                              double[] outreal, double[] outimag) {
```

```
    int n = inreal.Length;
    for (int k = 0; k < n; k++) {  //对于每个输出元素
        double sumreal = 0;
        double sumimag = 0;
        for (int t = 0; t < n; t++) {  //对于每个输入元素
            double angle = 2 * Math.PI * t * k / n;
            sumreal += inreal[t] * Math.Cos(angle) + inimag[t] * Math.Sin(angle);
            sumimag += -inreal[t] * Math.Sin(angle) + inimag[t] * Math.Cos(angle);
        }
        outreal[k] = sumreal;
        outimag[k] = sumimag;
    }
}
```

2.6 移除静音

用户可以从 WAV 文件中的开始和结束部分移除指定时间长度的音频段。例如，如果音频文件播放时长为1min，用户想要将该文件修剪成从20s～40s并保存成新文件，实现代码如下：

```
public static class WavFileUtils
{
    public static void TrimWavFile(string inPath,  //输入路径
    string outPath,                                //输出路径
    TimeSpan cutFromStart,                         //开始部分的时间长度
    TimeSpan cutFromEnd)                           //结束部分的时间长度
    {
        using (WaveFileReader reader = new WaveFileReader(inPath))
                                                   //读入文件
        {
            using (WaveFileWriter writer =
            new WaveFileWriter(outPath, reader.WaveFormat))  //写出文件
            {
                int bytesPerMillisecond =
                reader.WaveFormat.AverageBytesPerSecond/1000;
```

```
                //开始位置
                int startPos = (int)cutFromStart.TotalMilliseconds * bytesPerMillisecond;
                startPos = startPos - startPos % reader.WaveFormat.BlockAlign;
                int endBytes =
                (int)cutFromEnd.TotalMilliseconds * bytesPerMillisecond;
                endBytes = endBytes - endBytes % reader.WaveFormat.BlockAlign;
                //结束位置
                int endPos = (int)reader.Length - endBytes;
                TrimWavFile(reader, writer, startPos, endPos);
            }
        }
    }

    private static void TrimWavFile(WaveFileReader reader, WaveFileWriter writer,
                            int startPos, int endPos)
    {
        reader.Position = startPos;
        byte[] buffer = new byte[1024];
        while (reader.Position < endPos)
        {
            int bytesRequired = (int)(endPos - reader.Position);
            if (bytesRequired > 0)
            {
                int bytesToRead = Math.Min(bytesRequired, buffer.Length);
                int bytesRead = reader.Read(buffer, 0, bytesToRead);
                if (bytesRead > 0)
                {
                    writer.WriteData(buffer, 0, bytesRead);
                }
            }
        }
    }
```

检测音频文件中的静音，以便随后报告或截断。

以下为 AudioFileReader 类写了一个扩展方法，该方法返回文件开始/结尾处的静音持续时间。

```
    static class AudioFileReaderExt
```

```
{
    public enum SilenceLocation { Start, End } //是从开始处还是从结尾处判断静音
    //根据振幅判断是否静音
    private static bool IsSilence(float amplitude, sbyte threshold)
    {
        double dB = 20 * Math.Log10(Math.Abs(amplitude));
        return dB < threshold;
    }
    //获得静音持续时间
    public static TimeSpan GetSilenceDuration(this AudioFileReader reader,
                                    SilenceLocation location,
                                    sbyte silenceThreshold = -40)
    {
        int counter = 0;
        bool volumeFound = false;
        bool eof = false;
        long oldPosition = reader.Position;

        var buffer = new float[reader.WaveFormat.SampleRate * 4];
        while (!volumeFound && !eof)
        {
            int samplesRead = reader.Read(buffer, 0, buffer.Length);
            if (samplesRead == 0)
                eof = true;
            for (int n = 0; n < samplesRead; n++)
            {
                if (IsSilence(buffer[n], silenceThreshold))
                {
                    counter++;
                }
                else
                {
                    if (location == SilenceLocation.Start)
                    {
                        volumeFound = true;
                        break;
                    }
```

```
                else if (location == SilenceLocation.End)
                {
                    counter = 0;
                }
            }
        }
    }

    //重置位置
    reader.Position = oldPosition;
    double silenceSamples = (double)counter / reader.WaveFormat.Channels;
    double silenceDuration = (silenceSamples / reader.WaveFormat.SampleRate)
     * 1000;
    return TimeSpan.FromMilliseconds(silenceDuration);
  }
}
```

设置可接受几乎任意格式的音频文件，而不仅仅是.wav 格式文件，实现代码如下：

```
using (AudioFileReader reader = new AudioFileReader(filePath))
{
    //从开始位置的静默时间
    TimeSpan duration =
       reader.GetSilenceDuration(AudioFileReaderExt.SilenceLocation.Start);
    Console.WriteLine(duration.TotalMilliseconds);
}
```

第 3 章 Perl 开发语音识别

本章结合 Kaldi 中的相关代码来讲解 Perl 基本语法。

3.1 变量

Perl 中的变量不用声明，可以直接使用。但是声明变量有助于查错，是一种良好的编程习惯。Perl 中有 3 种变量用不同的前缀区分。

- $：标注标量。标量又有数字、字符串等类型，但这些类型可以自动互相转换。
- @：标注数组。
- %：标注 hash，又称关联数组。

数组的例子：

```
@ARGV < 2 && die "usage: run.pl log-file command-line arguments...";
```

3.1.1 数字

数字是一种标量，所以数字类型的变量以“$”开头。例如：

```
$jobstart = 1;
```

比较数字是否相等：

```
$val == 2
```

整数转换成字符串：

```
printf '%4d',2000;
```

浮点数转换成整数：

```
int($float)
```

浮点数转换成字符串：

```
printf '%e', $floatnum
```

3.1.2 字符串

字符串可以使用单引号或双引号表示。例如：

```
die "run.pl: Error waiting for child process";
```

单引号形式只会按照原样输出字符串，不会考虑其中的变量、转义符等情况；双引号形式则会解析其中的变量。例如，打印出字符串变量的内容：

```
my $localScalar = "This is a scalar variable.";
print("\$localScalar: $localScalar\n");
```

输出：

```
$localScalar: This is a scalar variable.
```

打印出数字变量的内容：

```
my $num = 4040.5;
print("$num\n");                              #4040.5
```

用点号连接字符串：

```
my $world = "World";
print("Hello ".$world."\n");                  #Hello world
```

如下代码：

```
$factoid = "$name lives at $address!";
```

等价于：

```
$factoid = $name . ' lives at ' . $address . '!';
```

连接传递过去的多个参数：

```
print("Hello ",$world,"\n");                          #Hello world
```

Perl 还提供了两个函数 q 和 qq 引用字符串。q()函数的功能和单引号类似，qq()函数的功能和双引号类似。这两个函数的主要目的是使用户不用转义符就能在字符串中使用单/双引号。

3.1.3 数组

定义一个数组并打印其中的元素。

```
my @array = (
   "print",
   "these",
   "strings",
   "out",
   "for",
   "me",                                         #结尾的逗号是允许的
);
#打印数组元素
print($array[0]);                                #输出 print
print($array[1]);                                #输出 these
print($array[2]);                                #输出 strings
print($array[3]);                                #输出 out
print($array[4]);                                #输出 for
print($array[5]);                                #输出 me
print($array[6]);                                #告警
print("\n");
```

使用 scalar()方法返回数组大小。

```
@names = (Jo, Pete, Bill, Bob, Zeke, Al);
print scalar(@names);   #6
```

3.1.4 散列表

声明一个散列表：

```
%scientists =();
```

给散列表赋初始值：

```
my %scientists = (
   "Newton"   => "Isaac",
   "Einstein" => "Albert",
   "Darwin"   => "Charles",
);

print($scientists{"Newton"}."\n");                  #Isaac
print($scientists{"Einstein"}."\n");                #Albert
print($scientists{"Darwin"}."\n");                  #Charles
```

使用 Data:Dumper 检查其中的内容。

```
use Data::Dumper;
print(Dumper(\%scientists) . "\n");
```

输出：

```
$VAR1 = {
   'Newton' => 'Isaac',
   'Einstein' => 'Albert',
   'Darwin' => 'Charles'
};
```

当数据结构有好几个层次时，以下的设置可以使输出更紧凑，可读性更高。

```
$Data::Dumper::Indent = 1;
```

使用 keys()函数返回一个键值的数组。

```
$ranks{"UCLA"} = 1;
$ranks{"OSU"} = 2;
@teams = keys (%ranks);
#@teams 是("UCLA","OSU"), 也有可能是("OSU","UCLA")
```

使用 each()函数返回一个“关键字-值”对。随后的调用返回剩下的“关键字-值”对，可用该函数来遍历散列表。

```
$ranks{"UCLA"} = 1;
$ranks{"OSU"} = 2;
while(($team, $rank) = each (%ranks)) {
   print("Ranking for $team is $rank\n");
```

```
}
```

使用 delete()函数从散列表删除一个“关键字-值”对，返回被删除元素的值。

```
$ranks{"UCLA"} = 1;
$ranks{"OSU"} = 2;
$x = delete $ranks{"UCLA"}; #现在%ranks 仅剩一个"关键字-值"对了
print("$x \n"); # $x 是 1
```

使用 exists()函数判断该元素是否已被删除。

```
$ranks{"UCLA"} = 1;
$ranks{"OSU"} = 2;
print("存在\n") if exists $ranks{"UCLA"}; #打印出"存在"
```

3.2 多维数组

创建数组：

```
$abc=[["00","01"],["10","11"]];
```

使用数组：

```
$var=$abc->[1][1];
Print($var);  #打印出“11”
```

这里的$abc 是一个引用。

3.3 常量

使用 constant 编译指示允许定义的常量。例如，声明一个标量常量：

```
use constant PI => 4 * atan2(1, 1);
```

声明一个列表常量：

```
use constant WEEKDAYS => qw(
   Sunday Monday Tuesday Wednesday Thursday Friday Saturday
```

```
);
```

使用时必须加括号：

```
print "Today is", (WEEKDAYS)[ 1 ], ".\n";
```

声明一个散列常量：

```
use constant  WEEKABBR =>{
   Monday   => 'Mon',
   Tuesday   => 'Tue',
   Wednesday  => 'Wed',
   Thu  => 'Thursday',
   Fri   => 'Friday');
```

使用举例如下：

```
%abbr = WEEKABBR;
$day = 'Wednesday';
print "The abbrevaiation for $day is ", $abbr{$day};
```

3.4　操作符

Perl 中的操作符细分，可分为赋值操作符、算术操作符、字符操作符、比较操作符、位操作符、逻辑操作符、组合赋值操作符、递增和递减操作符、正则表达式操作符、逗号操作符和关系操作符、引用操作符和访问引用操作符、箭头操作符、范围操作符、三元操作符、文件操作符、命令操作符。其中正则表达式操作符、文件操作符和命令操作符将在后面专门讲解。

把右边表达式的值赋予左边变量：

```
$x = 'value';
```

上述语句中，*x* 表示赋值操作符左边的实体。*x* 必须为变量，可以给它分配值。但不能向字符串赋值，如"constant"=132 这个语句就是错误的，因为常量字符串"constant"不能作为变量名。

Perl 中的字符串操作符包括连接操作符“.”和复制操作符“x”。

①连接操作符“.”相当于字符串间的加号。

```
$example = 'Hello '. 'World';
```

但是，如果实际使用语句为：

```
$example = 'Hello '+'World';
```

返回的结果将是 0。因为该语句执行的是整型运算，相加的字符串会被转换成整数 0，再相加。

②复制操作符“x”左边是一个字符串或一个列表，右边代表左边元素复制的次数。例如：

```
$example = "\t" x 8;
@array = (1,2,3,4,5) x 2;   # @array =(1,2,3,4,5,1,2,3,4,5);
```

最简单、最常用的比较操作符多用于测试一个值是否等于另一个值。如果值相等，则测试返回 true；如果值不相等，则测试返回 false。数字和字符串有不同的比较操作符。测试两个数值的相等性，可以使用比较操作符“==”。测试两个字符串值的相等性，可以使用比较操作符 eq(EQual)。

以下是两个例子。

```
if (5 == 5) { print "== for numeric values\n"; }
if ('moe' eq 'moe') { print "eq (EQual) for string values\n"; }
```

数字和字符串的比较操作符如表 3-1 所示。

表 3-1　数字和字符串的比较操作符及说明

数字比较操作符	字符串比较操作符	说　明
<	lt	小于
>	gt	大于
==	eq	等于
<=	le	小于或等于
>=	ge	大于或等于
!=	ne	不等于
<=>	cmp	比较两个值的大小。当两个值相等时，返回 0；当第一个值大时，返回 1；当第二个值大时，返回 –1

组合操作符如表 3-2 所示。

表 3-2　组合操作符的类型及说明

操　作　符	操作符类型	说　　明
+=	算术操作符	相加并赋值。$x += $y；等价于 $x = $x + $y;
−=	算术操作符	相减并赋值。$x −= $y；等价于 $x = $x−$y;
*=	算术操作符	相乘并赋值。$x *= $y；等价于 $x = $x * $y;
/=	算术操作符	相除并赋值。$x /= $y；等价于 $x = $x / $y;
%=	算术操作符	取余并赋值。$x %= $y；等价于 $x = $x % $y;
**=	算术操作符	乘幂并赋值。$x **= $y；等价于 $x = $x ** $y;
x=	字符串操作符	重复并赋值。$x x= 3；等价于 $x = $x x 3;
.=	字符串操作符	连接并赋值。$x .= $y；等价于 $x = $x . $y;
<<=	移位操作符	左移并赋值。$x <<= $y；等价于 $x = $x << $y;
>>=	移位操作符	右移并赋值。$x <<= $y；等价于 $x = $x << $y;
&&	逻辑操作符	逻辑与并赋值。$x &&= $y；等价于 $x = $x && $y;
||=	逻辑操作符	逻辑或并赋值。$x ||= $y；等价于 $x = $x || $y;
|=	位操作符	位或并赋值。$x |= $y；等价于 $x = $x | $y;
&=	位操作符	位与并赋值。$x &= $y；等价于 $x = $x & $y;
^=	位操作符	位异或并赋值。$x ^= $y；等价于 $x = $x ^ $y;

尝试用下面的例子来理解 Perl 中可用的所有赋值操作符。

```
#!/usr/local/bin/perl
$a = 10;
$b = 20;
print "Value of \$a = $a and value of \$b = $b\n";
$c = $a + $b;
print "After assignment value of \$c = $c\n";
$c += $a;
print "Value of \$c = $c after statement \$c += \$a\n";
$c -= $a;
```

```
print "Value of \$c = $c after statement \$c -= \$a\n";
$c *= $a;
print "Value of \$c = $c after statement \$c *= \$a\n";
$c /= $a;
print "Value of \$c = $c after statement \$c /= \$a\n";
$c %= $a;
print "Value of \$c = $c after statement \$c %= \$a\n";
$c = 2;
$a = 4;
print "Value of \$a = $a and value of \$c = $c\n";
$c **= $a;
print "Value of \$c = $c after statement \$c **= \$a\n";
```

执行上面的代码会产生以下结果：

```
Value of $a = 10 and value of $b = 20
After assignment value of $c = 30
Value of $c = 40 after statement $c += $a
Value of $c = 30 after statement $c -= $a
Value of $c = 300 after statement $c *= $a
Value of $c = 30 after statement $c /= $a
Value of $c = 0 after statement $c %= $a
Value of $a = 4 and value of $c = 2
Value of $c = 16 after statement $c **= $a
```

3.5 控制流

for 语句是能够实现自动增加循环变量的循环结构。它的语法结构如下：

```
for (初始化表达式; 测试表达式; 增加循环变量)
{代码块}
```

当 Perl 遇到一个 for 循环时，执行顺序如下。

- 初始化表达式被计算。
- 测试表达式被计算。如果它的计算结果为真，代码块就运行。
- 当该代码块执行结束后，便执行递增操作，并再次计算测试表达式。如果该测

试表达式的计算结果仍然为真，那么代码块再次运行。这个进程将继续下去，直到测试表达式的计算结果为假为止。

示例代码如下：

```
$jobstart = 1;
$jobend = 10;
for ($jobid = $jobstart; $jobid <= $jobend; $jobid++) {
    print $jobid . "\n";
}
```

3.6　文件与目录

Perl 程序是通过文件句柄来进行 I/O 操作的。特别地，Perl 提供了默认的文件句柄 STDIN（代表标准输入）、STDOUT（代表标准输出）和 STDERR（代表标准错误输出）。STDERR 输出错误信息的语句如下：

```
    print STDERR "run.pl: Warning: failed to detect any processors from /proc/
cpuinfo\n";
```

写文件的例子，实现代码如下：

```
$append = 0;
if ($append)
{
   open(MYOUTFILE, ">filename.out");      #覆盖写
}
else
{
   open(MYOUTFILE, ">>filename.out");     #追加写
}
print MYOUTFILE "Timestamp: ";            #写文本,没有换行
print MYOUTFILE timestamp();              #把函数返回值写到文本
print MYOUTFILE "\n";                     #写换行符
#*** 打印自由文本,需要后面的分号***
print MYOUTFILE <<"MyLabel";
Steve was here
and now is gone
```

```
but left his name
to carry on.
MyLabel
close(MYOUTFILE); #关闭文件
```

3.7 例程

例程（Subroutine）又称函数，是结构化程序设计的基础。它接受多个输入参数，返回一个输出参数。

定义语法：

```
sub Subroutine name[()]
{
   sequence of statements;
}
```

用以下语句定义一个简单的函数，然后调用它。因为 Perl 在执行程序前会先编译程序，所以声明例程的位置并不重要。

```
#!/usr/bin/perl
#定义函数
sub Hello{
   print "Hello, World!\n";
}
#调用函数
Hello();
```

当上述的程序执行后，会输出以下结果。

```
Hello, World!
```

传递参数给例程是通过特殊变量“@_”完成的。例如：

```
sub Logger($)
{
   ($line) = @_;
   print "$line\n";
}
```

以下是通过 shift()函数访问输入参数的例子。

```
sub Logger($)
{
   $line = shift;
    print "$line\n";
}
```

3.8　执行命令

执行操作系统命令常用的方法是调用 system()函数和用反小点引号操作符 qx。以下是使用 system()函数的例子。

```
my $ret = system($cmd);
if ($ret!=0 )
{
   die "error during execute $cmd\n extend_error=$^E \nerrorno=$!\n ";
}
```

打开日志文件，并把结果输出到文件中。

```
$logfile = "test.txt";
open(F, ">$logfile") || die "run.pl: Error opening log file $logfile";
print F "#Started at " . 'date';
```

3.9　正则表达式

正则表达式是一个描述模式（pattern）的字符串。在 Perl 中，正则表达式用来查找/替换字符串、提取字符串中想要的部分等。

3.9.1　基本类型

最简单的正则表达式是匹配一个单词，例如：

```
"Hello World" =~ /World/;  #匹配上了
```

这里“"Hello World"”是一个字符串；“World”是正则表达式，而“/World/”是为了告知 Perl 匹配搜索字符串；操作符“=~”连接字符串和正则表达式匹配。如果正则表达式匹配成功，整个表达式返回 true，否则返回 false。在这个例子中，“World”匹配上了字符串“"Hello World"”，因此表达式返回值为 true。如同下例这样可以把该表达式用在 if 条件判断中。

```
if ("Hello World" =~ /World/) {
   print "It matches\n";
}
else
{
   print "It doesn't match\n";
}
```

3.9.2 正则表达式模式

模式字符串中存在一些特殊的字符串。下面逐一讲解这些字符串。

“^”匹配字符串的开始（或匹配行的开始，如果使用“/m”）。例如：

```
$var =~ s/^\s+//;    #左边的 trim
```

“$”匹配字符串的结束（或匹配行的结束，如果使用“/m”）。例如：

```
$var =~ s/\s+$//;    #右边的 trim
```

转义字符“\”用来转义随后的一个字符。“\$”不再代表结束符，而代表“$”。例如，模式：

```
/a\.b/
```

匹配：

```
"proga.bat"
```

不匹配：

```
"a..b"
```

转义字符及说明如表 3-3 所示。

表 3-3 转义字符表

转义字符	说 明	转义字符	说 明
\t	tab	\x1B	十六进制数字符
\n	新行	\c[	控制字符
\r	回车	\l	小写下一个字符
\f	进纸	\u	大写下一个字符
\a	警告	\L	小写一直到\E
\e	Esc	\U	大写一直到\E
\033	八进制数字符	\E	结束大小写修改

“.” 匹配任何单个字符，除了一个新行（除非使用了“/s”）。

例如，模式：

```
/f./
```

匹配：

```
"this is fun"
```

不匹配：

```
"nothing beyond the f"
```

“*” 匹配前面的元素 0 次或多次。

```
$a="(12)34(56)78(90)abc";
$a=~s/\(.*\)//g;
print"$a\n";   #输出 abc
```

默认情况下“*”是贪婪的。要让它不贪婪，需要在后面加上“?”。

```
$a="(12)34(56)78(90)abc";
$a=~s/\(.*?\)//g;
print"$a\n"; #输出 3478abc
```

“+”匹配前面的元素 1 次或多次。例如，“/\d+/”匹配一个无符号整数。和“*”一样，默认情况下“+”是贪婪的。要让它不贪婪，需要在后面加上“?”。

“?”匹配前面的元素 0 次或 1 次。特殊地，“?”可以用来修饰“*”。

3.10 命令行参数

使用 use warnings;可帮助用户发现程序中的输入错误，例如少输入了分号、使用'elseif'而不是使用'elsif'，或者正在使用废弃的语法、函数。注意：使用警告只会提供警告并继续执行，即不会中止执行。

除了使用 warnings 以外，也可以使用-w。但是它们的作用范围不一样。

- warnings 起作用的范围是包含它的块，而-w 的作用范围却是全局的。
- 使用 warnings 可以有选择地打开某些警告，或是有选择地关闭某些警告，但使用-w 要么是打开所有可选的警告，要么是根本没有可选的警告。推荐使用 use warnings;。

第 4 章 Python 开发语音识别

从 Python 官方网站（https://www.python.org/downloads/）下载操作系统对应的安装包，在同一台计算机上安装 Python 的多个版本。为了配合 Kaldi 的开发，可以安装 Python 2.7 和 Python 3.6 两个版本，并且把 Python 2.7 作为默认版本。

4.1 Windows 操作系统下安装 Python

在图形化用户界面出现前，人们就是用命令行来操作计算机的。Windows 命令行是通过 Windows 操作系统目录下的 cmd.exe 执行的。执行该程序最直接的方式是找到该程序，然后双击。但 cmd.exe 并没有一个桌面的快捷方式，操作起来太麻烦。我们需要在“开始”菜单的“运行”窗口中直接输入程序名，按 Enter 键后运行该程序。选择“开始”→“运行”命令，或按 Windows+R 组合键，这样就会打开资源管理器中的“运行”窗口。总之，输入程序名“cmd”后单击“确定”按钮，出现命令提示窗口。因为能够通过这个黑屏的窗口直接输入命令来控制计算机，所以该窗口也称为控制台窗口。

正如公园的地图上往往会标出游客的当前位置一样，Windows 命令行也有个当前路径的概念。C:\Users\Administrator 就是当前路径。用 cd 命令可以改变当前路径，例如改变到 C:\Python\Python27 路径，可以使用如下命令。

```
C:\Users\Administrator>cd C:\Python\Python27
```

如果输入“cd d:”，这样是改变当前路径到D:根目录。所以切换盘符不能使用cd命令，而是直接输入盘符的名称。例如想要切换到D盘，可以使用如下命令。

```
C:\Users\Administrator>d:
```

系统约定从指定的路径找可执行文件。这个路径通过PATH环境变量指定。环境变量是一个“变量名=变量值”的对应关系，每一个变量都有一个或者个值与之对应。如果是多个值，则这些值之间用“;”分开。例如，PATH环境变量可能对应值为“C:\Windows\system32;C:\Windows”，表示Windows会从C:\Windows\system32和C:\Windows两个路径找可执行文件。

设置或修改环境变量的具体操作步骤：首先在Windows桌面上用鼠标右键单击“我的电脑”图标，选择“属性”→“高级”→“环境变量”，然后设置用户变量或系统变量，再设置环境变量PATH的值。

注意：如果是用Windows 2008以后的操作系统，可能找不到“我的电脑”这样的快捷图标。其实打开桌面上的“我的电脑”就是运行资源管理器。打开资源管理器的另一种方法是：按住键盘上的Windows键不放，再按E键。

需要重新启动命令行才能让环境变量设置生效。为了检查环境变量是否设置正确，可以在命令行中显示指定环境变量的值，需要用到echo命令。echo命令用来显示一段文字：

```
C:\Users\Administrator>echo Hello
```

执行上述命令将在命令行输出：Hello。

如果要引用环境变量的值，可以用前后两个百分号把变量名包围起来，即“%变量名%”。echo命令用来显示一个环境变量中的值：

```
C:\Users\Administrator>echo %PATH%
```

此外，也可以在命令行直接输入“PATH”来显示这个环境变量的值。

假设把Python安装在D:\Python\Python27目录下，则可以在计算机属性中手工设置PATH环境变量，然后检查环境变量的值：

```
C:\Users\Administrator.PC-201509301458>echo %PATH%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\Syste
m32\WindowsPowershell\v1.0\;D:\apache-maven-3.5.2\bin;D:\Python\Python27
```

使用 SETX 命令设置环境变量。用 SETX 命令设置的环境变量可以保证重启后一样有用。这里增加 Python 的安装目录到 PATH 环境变量：

```
SETX PATH "%PATH%;D:\Python\Python27"
```

更专业的方式是开发一个软件包管理工具，可以直接输入安装包的名称来安装它。

以下命令检查 Python 是否正确安装，以及所使用的版本号。

```
>python -version
```

4.2　Linux 操作系统下安装 Python

检查 Python 3 是否已经正确安装及其版本号：

```
#python 3 -V
Python 3.4.5
```

检查 Python 3 所在的路径：

```
#which python 3
/usr/bin/python 3
```

如果使用 CentOS，可以使用 YUM 安装 Python 3。使用 YUM 命令查找可供安装的 Python 版本：

```
#yum search python 3
```

安装想要的版本：

```
#yum install python 36
```

如果使用 Ubuntu 操作系统，执行以下命令可以更新软件包列表，并将所有系统软件升级到可用的最新版本：

```
#sudo apt-get update && sudo apt-get -y upgrade
```

安装 Pip 包管理系统：

```
#sudo apt-get install python 3-pip
```

4.3 选择版本

Linux 操作系统中有可能同时存在多个可用的 Python 版本。每个 Python 版本都对应一个可执行二进制文件，可以使用 ls 命令来查看系统中有哪些 Python 的二进制文件可供使用。

```
$ls /usr/bin/python*
```

使用 python 命令可以执行 Python 2，使用 python 3 命令可以执行 Python 3。那么，如何使用 python 命令执行 Python 3 呢？一种简单、安全的方法是使用别名，将如下命令放入~/.bashrc 或~/.bash_aliases 文件中。

```
alias python=python 3
```

最好在终端中使用'python 3'命令，在 Python 3.x 文件中使用 shebang 行'#!/usr/bin/env python3'。

4.4 开发环境

关于开发环境，既可以使用 Sublime 这样的简单文本编辑器编写 Python 代码，也可以使用 PyCharm 这样的专业集成开发环境。

PyCharm 是业界公认的 Python 开发工具之一，尤其在智能代码助手、代码自动提示、重构、GIT 整合、代码审查、创新的 GUI 设计等方面的功能可以说是超常的。PyCharm 是 JetBrains 公司的产品，这家公司总部位于捷克共和国的首都布拉格，开发人员以严谨著称的东欧程序员为主。用于 Java 开发的 IDEA 是这家公司的主力产品，PyCharm 的大部分功能都可以通过免费的 Python 插件提供给 IDEA。有两个版本的 PyCharm：

专业版（免费 30 天试用版）和功能较少的社区版本（Apache 2.0 许可证）。

4.5　注释

和 shell 类似，Python 脚本中用“#”表示注释。但如果“#”位于第一行开头，并且是“#!”（称为 Shebang）则例外，它表示该脚本使用后面指定的解释器/usr/bin/python3 解释执行。每个脚本程序只能在开头包含这个语句。

为了能够在源代码中添加中文注释，需要把源代码保存成 utf-8 格式的。例如：

```
#-*- coding: utf-8 -*
import tensorflow as tf
x = tf.placeholder("float", [2])  #形状是 2
```

4.6　变量

定义变量时不声明类型，但变量在内部是有类型的。使用函数 type()获得变量的类型，实现代码如下：

```
>>> a='3'
>>> type(a)
<class 'str'>
```

4.6.1　数值

Python 中有 int（整数）、float（浮点数）和 complex（复数）3 种不同的数值类型。与 Java 或 C 语言中的 int 类型不同，Python 语言中的 int 类型是无限精度的。例如：

```
>>> i=32432444444444444444444444444444444444444444448797687567567657000000000000
0000000000000000000000000000000000000000000000000000000000000000000564564
>>> i
3243244444444444444444444444444444444444444444879768756756765700000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000564564
    >>> type(i)
    <class 'int'>
```

Python 依据 IEEE 754 标准使用二进制表示浮点数（float），存在表示精度的问题。例如：

```
>>> 0.1 == 0.1000000000000000000000001
True
```

使用 decimal 模块可以用十进制表示完整的小数。例如：

```
>>> import decimal
>>> a = decimal.Decimal('0.1')
>>> b = decimal.Decimal('0.100000000000000000000001')
>>> a == b
False
```

在傅里叶变换中会用到复数。复数在 Python 中是一个基本数据类型。例如：

```
>>> complex(2,3)
(2+3j)
```

一个复数有一些内置的访问器。例如：

```
>>> z = 2+3j
>>> z.real
2.0
>>> z.imag
3.0
>>> z.conjugate()
(2-3j)
```

内置函数 abs()和 pow()支持复数。例如：

```
>>> abs(3 + 4j)
5.0
>>> pow(3 + 4j, 2)
(-7+24j)
```

标准模块 cmath 具有处理复数的更多功能。例如：

```
>>> import cmath
```

```
>>> cmath.sin(2 + 3j)
(9.15449914691143-4.168906959966565j)
```

用于数值运算的算术操作符说明如表 4-1 所示。

表 4-1　算术操作符说明

操　作　符	程序表达式	数学表达式	说　明
+	a+b	$a+b$	加号
−	a−b	$a-b$	减号
*	a*b	$a\times b$	乘号
/	a/b	$a\div b$	除号
//	a//b	$\lfloor a\div b \rfloor$	取整除
%	a%b	$a \bmod b$	求模
−	−a	$-a$	取负数
abs()	abs(a)	$\|a\|$	求绝对值
**	a**b	a^b	求指数
sqrt()	math.sqrt(a)	$\sqrt{a}$	求平方根

对于“/”运算，就算分子和分母都是 int 类型，返回的也将是浮点数。例如：

```
>>> print(1/3)
0.3333333333333333
```

Python 支持不同类型的数据相加，它使用数据类型强制转换的方式来解决数据类型不一致的问题——将一个操作数转换成与另一个操作数相同的数据类型。例如，将整数转换为浮点数，非复数转换为复数。

4.6.2　字符串

使用 strip()方法可以去掉字符串首尾的空格或指定的字符。

```
term = "   hi      ";   #去除首尾空格
print(term.strip());
```

4.7 数组

使用 array（数组）可以存储同一类型的数据。通过 import array 命令导入 Python 的数组类型，就可以使用 array 类型了。例如：

```
from array import array
node=array('H')   #存储无符号短整型的数组
node.append(12)
```

4.8 列表

使用 list 可以存储任何类型的对象。例如：

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print("list1[0]: ", list1[0])
print("list2[1:5]: ", list2[1:5])
```

输出：

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

4.9 元组

元组是一个不可变的 Python 对象序列。元组变量的赋值要在定义时就进行，赋值后不允许有修改。例如：

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print( "tup1[0]: ", tup1[0]);
print( "tup2[1:5]: ", tup2[1:5]);
```

4.10　字典

字典是另一种可变容器模型，且可存储任意类型对象。要访问字典元素，可以使用熟悉的方括号和键来获取它的值。

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

4.11　控制流

控制流用来根据运行时情况调整语句的执行顺序。流程控制语句可以分为条件语句和迭代语句。

4.11.1　条件判断

当路径不存在时，就创建它——使用条件语句实现。条件语句的一般形式如下：

```
if 条件:
    语句 1
    语句 2
    …
elif 条件:
    语句 1
    语句 2
    …
else:
    语句 1
    语句 2
    …
语句 x
```

例如，判断一个数是否为正数。

```
x = -32.2;
isPositive = (x > 0);
if isPositive:
    print(x, " 是正数");
else:
print(x, " 不是正数");
```

使用关系运算符和条件运算符作为判断依据。关系运算符返回一个布尔值，关系运算符的说明如表 4-2 所示。

表 4-2　关系运算符说明

运算符说明	用 法 说 明	返回 true，如果…
>	a > b	a 大于 b
>=	a >= b	a 大于或等于 b
<	a < b	a 小于 b
<=	a <= b	a 小于或等于 b
==	a == b	a 等于 b
!=	a != b	a 不等于 b

4.11.2　循环

使用复印机复印一个证件，可以设置复制的份数。在 Python 中，可以使用 for 循环或 while 循环实现多次重复执行一个代码块。

for 循环可以遍历任何序列。例如，列出当前目录下所有的文件。

```
import glob
cur_files = glob.glob("*");
for wf in cur files:
print(wf)
```

每一次在执行循环代码块前，根据循环条件决定是否继续执行循环代码块，当满足循环条件时，继续执行循环体中的代码。在循环条件之前写上关键词 while（这里的 while 就是“当”的意思），当用户直接按 Enter 键时退出循环。例如：

```
import sys
while True:
    line = sys.stdin.readline().strip()
    if not line:
        break
    print(line)
```

4.12　模块

使用 import 语句可以导入一个.py 文件中定义的函数。一个.py 文件就称为一个模块（Module）。例如，存在一个 re.py 文件，可以使用 import re 语句导入这个正则表达式模块。

使用正则表达式模块去掉一些标点符号的例子，实现代码如下：

```
import re
line = 'Hi.'
normtext = re.sub(r'[\.,:;\?]', '', line)
print(normtext)
```

从 re 模块直接导入函数 sub()的例子，实现代码如下：

```
from re import sub
line = 'Hi.'
normtext = sub(r'[\.,:;\?]', '', line)
print(normtext)
```

模块越来越多以后，会难以管理——可能会出现重名的模块。例如，一个班里有两个叫陈晨的同学，如果他们在不同的小组，则可以分别称为第一组的陈晨和第三组的陈晨，这样就能区分同名了。为了避免名称冲突，可以让模块位于不同的命名空间——包中，在模块名前面加上包名限定。这样即使模块名相同，也不会冲突了。

查看已经安装的模块：

```
D:\Python\Python36\Tools\scripts>..\..\python pydoc3.py modules
```

刚开始时，setup.py 文件可能令人望而生畏。使用 twine 包可以把模块发布共享到

pypi 网站（https://pypi.org/）。例如，TensorFlow 模块位于 https://pypi.python.org/pypi/tensorflow。wheel 本质上是一个 zip 包格式，它使用.whl 扩展名，用于 Python 模块的安装。pip 提供了一个 wheel 子命令来安装 wheel 包。当然，需要先安装 wheel 模块。

```
pip install wheel
python setup.py  bdist_wheel
```

在 Windows 操作系统下使用 wheel 文件安装 numpy。首先下载用于 Windows 操作系统的 Python 扩展包 numpy-1.15.0rc1+mkl-cp35-cp35m-win_amd64.whl。然后在命令行输入：

```
C:\Users\Downloads> pip install numpy-1.15.0rc1+mkl-cp35-cp35m-win_amd64.whl
```

4.13　函数

把一段多次重复出现的代码命名成一个有意义的名称，然后通过调用该名称来执行这段代码。这种有名称的代码段就是函数。例如，定义一个名为 RunKaldiCommand 的函数。

```
import subprocess
def RunKaldiCommand(command, wait = True):
    """通常执行由管道连接的一系列命令,所以我们使用 shell=True """
    p = subprocess.Popen(command, shell = True,
                         stdout = subprocess.PIPE,
                         stderr = subprocess.PIPE)

    if wait:
        [stdout, stderr] = p.communicate()
        if p.returncode is not 0:
            raise Exception("There was an error while running the command {0}\n".format(command)+"-"*10+"\n"+stderr)
        return stdout, stderr
    else:
        return p
```

使用该函数：

```
RunKaldiCommand("ls -lh")
```

这里只给 RunKaldiCommand()的第一个参数传递了值，第二个参数的值采用默认的 True。

每个 Python 文件/脚本（模块）都有一些未明确声明的内部属性。其中一个属性是_builtins_属性，它本身包含许多有用的属性和功能。我们可以在这里找到_name_属性，根据模块的使用方式，它可以具有不同的值。当把 Python 模块作为程序直接运行时（无论是从命令行还是双击它），_name_中包含的值都是文本字符串"_main_"。相比之下，当一个模块被导入另一个模块（或在 python REPL 被导入）时，_name_属性中的值是模块本身的名称（即隐式声明它的 Python 文件/脚本名称）。

Python 脚本执行的方式是自上而下的，指令在解释器读取时才执行。如果你想要做的只是导入模块并利用它的一个或两个方法，这可能是一个问题。那么，应该怎么做呢？你可以有条件地执行这些指令——将它们包装在一个 if 语句块中。这是 main()函数的目的。它是一个条件块，因此除非满足给定的条件，否则不会处理 main()函数。

Main()函数的例子，实现代码如下：

```
import sys

def main():
    if len(sys.argv) != 2:
        sys.stderr.write("Usage: {0} <min-count>\n".format(sys.argv[0]))
        raise SystemExit(1)

    words = {}
    for line in sys.stdin.readlines():
        parts = line.strip().split()
        words[parts[1]] = words.get(parts[1], 0) + int(parts[0])

    for word, count in words.iteritems():
        if count >= int(sys.argv[1]):
            print ("{0} {1}".format(count, word))

if  name  == ' main ':
main()
```

4.14 读写文件

逐行读入文本文件：

```
lexicon = open("lexicon.txt")
for line in lexicon:
    line = line.strip()
    print(line,"\n")
lexicon.close()
```

读入 utf8 编码格式的文本文件：

```
import codecs
import sys
transcript = codecs.open(sys.argv[1], "r", "utf8")    #第一个参数传入文件名
for line in transcript:
    print(line)
transcript.close()
```

为了实现写入文本文件，可以使用'w'模式的函数 open()以写模式打开新文件。

```
new path = "a.speaker info"
fout = open(new_path,'w')
```

需要注意的是，如果 a.speaker_info 在打开文件之前已经存在，它的旧内容将被破坏，所以在使用'w'模式时要小心。

一旦打开新文件，就可以使用写入操作<file>.write()将数据放入文件中。写入操作接受单个参数，该参数必须是字符串，并将该字符串写入文件。如果想要在文件中开始新行，则必须明确提供换行符。例如：

```
fout.write("\nID:\t1212")
```

关闭文件可确保磁盘上的文件和文件变量之间的连接已完成，还可确保其他程序能够访问它们并保证数据安全，所以一定要确保关闭文件。现在，让我们使用<file>.close()关闭所有文件。

```
fout.close()
```

导入 json 模块读取 json 格式的文件：

```
import json
data = json.load(open('my_file.json', 'r'))
```

json 文件的内容如下：

```
{"hello":"lietu"}
```

读取 json 格式的文件如下：

```
>>> import json
>>> print(json.load(open('my file.json','r')))
{u'hello': u'lietu'}
```

4.15　面向对象编程

语音识别软件往往由很多万行代码组成，是一个复杂的系统。为了能够封装细节，需要为其抽象出对象。只要写出对象的实现代码，就可以创建出该对象并使用它。

例如，定义一个封装表中数据的 Table 类。

```
class Table(object):
    def  init (self, data=[], colnames=[]):
        self.data = data
        self.colnames = colnames
        self.colSep = '\t'
        self.lineSep = '\n'
    def data2str(self):
        strdata = []
        for r in self.data:
            strdata.append([str(c) for c in r])
        return strdata
    def _str_(self):
        sd = self.data2str()
        colwidth = [len(c) for c in self.colnames]
        for j in range(len(colwidth)):
            for r in sd:
                colwidth[j] = max(colwidth[j], len(r[j]))
        gaps = [m - len(c) for (m, c) in zip(colwidth, self.colnames)]
```

```
        rows = [self.colSep.join(
            [c + ' ' * gap for c, gap in zip(self.colnames, gaps)])]
        for r in sd:
            gaps = [m - len(c) for (m, c) in zip(colwidth, r)]
            rows.append(
                self.colSep.join([c + ' ' * d for c, d in zip(r, gaps)]))
        return self.lineSep.join(rows)
```

方法与对象实例或类相关联，函数则不是。当 Python 调度（调用）一个方法时，它会将该调用的第一个参数绑定到相应的对象引用。对于大多数方法，该参数通常称为 self。

为了实现动态加载一个 Python 类，可以使用如下函数。

```
def my import(name):
    components = name.split('.')
    mod =  import (components[0])
    for comp in components[1:]:
        mod = getattr(mod, comp)
    return mod
```

简单的_import_不起作用的原因是任何经过包字符串中第一个点的任何导入都是正在导入模块的属性。因此，以下代码不会奏效。

```
_import_('foo.bar.baz.qux')
```

必须使用以下代码调用上面的函数。

```
klass = my import('my package.my module.my class')
some_object = klass()
```

4.16　命令行参数

在采用多种编程语言开发的语音识别系统中，Python 脚本可能需要从命令行直接读取参数。如果脚本很简单或临时使用，没有多个复杂的参数选项，则可以直接用 sys.argv 读取传入的命令行参数。

测试 TestArgv.py 中内容的代码如下：

```
import sys

print("This is the path of the script: ", sys.argv[0])  #脚本的相对路径
print("Number of arguments: ", len(sys.argv))#长度最少为 1
print("The arguments are: " , str(sys.argv)) #str 函数输出 sys.argv 的内容
```

输出结果如下：

```
D:\PycharmProjects\Scripts\python.exe D:/PycharmProjects/untitled/TestArgv.py
a b c
This is the path of the script:  D:/PycharmProjects/untitled/TestArgv.py
Number of arguments:  4
The arguments are:  ['D:/PycharmProjects/untitled/TestArgv.py', 'a', 'b', 'c']
```

相关的规范有 POSIX getopt()和 GNU getopt_long()，单字符选项（-a）、组合选项（-abc 等同于-a-b-c）、多字符选项（-inum）、带参数的选项（-a arg, -inum 3, -a=arg）。

GNU 扩展 getopt_long()可以解析更可读的多字符选项，该选项前缀为双破折号，而非单个破折号。双破折号选项（如--inum）可以和单个破折号选项（-abc）区分开。GNU 扩展允许带参选项有不同的形式：--name=arg。

argprase 包使这一工作变得简单而规范，它支持 POSIX getopt()和 GNU getopt_long()。例如，有两个必需的参数 i 和 o，分别用于指定输入和输出文件。TestArgprase.py 实现代码如下：

```
import argparse
parser = argparse.ArgumentParser(description='format acronyms from a. b. c.
to a b c')
parser.add argument('-i','--input', help='Input ctm file ',required=True)
parser.add argument('-o','--output',help='Output ctm file', required=True)
args = parser.parse args()

fin = open(args.input,"r")
fout = open(args.output,"w")
```

使用-h 参数运行 TestArgprase.py 的输出结果如下：

```
    D:\PycharmProjects\untitled\venv\Scripts\python.exe D:/PycharmProjects/
untitled/TestArgprase.py -h
    usage: TestArgprase.py [-h] -i INPUT -o OUTPUT
    format acronyms from a. b. c. to a b c
    optional arguments:
      -h, --help         show this help message and exit
      -i INPUT, --input INPUT
                         Input ctm file
      -o OUTPUT, --output OUTPUT
                         Output ctm file
```

4.17 数据库

使用 sqlite3 模块在内存中创建一个 SQLite 数据库。

```
    import sqlite3
    conn = sqlite3.connect(':memory:')   #连接数据库
    print("成功打开数据库") ;
    c = conn.cursor()
    c.execute(
    '''CREATE TABLE results (exp text, dataset text, lm text, lm w int, wer float,
ser float)''')
```

从前面创建的 results 表中获取并显示记录。

```
    #获得排好序的所有结果
    c.execute("SELECT * FROM results ORDER BY exp, dataset, lm, lm_w")
    d = c.fetchall()
    t = Table(data=d, colnames=['exp', 'set', 'lm', 'LMW', 'WER', 'SER'])
    print('%s\n==================' % str(t))
```

4.18 日志记录

机器学习的训练过程可能用时很长，为了监控运行状态，可以用日志记录。

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.debug('trainning...')
```

日志级别大小关系为 CRITICAL ＞ ERROR ＞ WARNING ＞ INFO ＞ DEBUG ＞ NOTSET，当然也可以自己定义日志级别。

处理器将日志记录发送到任何输出，这些输出以自己的方式处理日志记录。例如，FileHandler 可获取日志记录并将其附加到文件中。

标准日志记录模块已经配备了多个内置处理器。例如：

- TimeRotated、SizeRotated 和 Watched 可以写入文件的文件处理器。
- StreamHandler 可以输出到 stdout 或 stderr 等流。
- SMTPHandler 通过电子邮件发送日志记录。
- SocketHandler 将日志记录发送到流套接字。

此外，还有 SyslogHandler、NTEventHandler、HTTPHandler 和 MemoryHandler 等处理器。

格式器负责将元数据丰富的日志记录序列化为一个字符串。如果没有提供，则有一个默认格式器。记录库提供的通用格式器类将模板和样式作为输入，然后可以为日志记录对象中的所有属性声明占位符。举个例子，'%(asctime)s %(levelname)s %(name)s:%(message)s'会生成类似以下这样的日志。

```
2017-07-19 15:31:13,942 INFO parent.child: Hello EuroPython.
```

注意，属性消息是使用提供的参数对日志的原始模板进行插值计算的结果。例如，对于 logger.info("Hello %s", "Laszlo")，消息将是"Hello Laszlo"。

TestStreamHandler.py 的实现代码如下：

```
import logging
logger = logging.getLogger( name )
logger.setLevel(logging.INFO)
handler = logging.StreamHandler()
handler.setLevel(logging.INFO)
formatter = logging.Formatter('%(asctime)s [%(filename)s:%(lineno)s
- %(funcName)s - %(levelname)s ] %(message)s')
```

```
handler.setFormatter(formatter)
logger.addHandler(handler)
string = ''
logger.info("trainning... \n {0}".format(string))
```

输出结果如下：

```
2018-06-24 22:01:28,465 [TestStreamHandler.py:12 - <module> - INFO ] trainning...
```

4.19 异常处理

在 Python 中，可以在运行时可能出现问题的代码中检查是否有异常。在 try 代码块中捕捉异常，而在 except 代码块中处理异常，这样把异常处理代码和正常的流程分开，可以使正常的处理流程代码连贯在一起。

except 代码块又称为异常处理器，常见格式为：

```
except ExceptionClass as e:  //异常类型
```

以下代码用于捕捉路径创建中的异常。

```
import errno
output dir = "d:/test"
try:
       os.makedirs(output dir)
except OSError as e:
       if e.errno == errno.EEXIST and os.path.isdir(output dir):
          print("路径已经存在");
          pass
       else:
          raise e
```

4.20 测试

nose 是 Python 中的一个测试框架。安装 nose 后，运行对 numpy 的测试。例如：

```
python -c "import numpy;numpy.test()"
```

4.21　语音活动检测

语音活动检测（Voice Activity Detection，VAD）也称为语音检测，它是一种语音处理技术，用于检测是否存在人类语音。

（1）声学特征提取

声音是模拟信号，声音的时域波形只代表声压随时间变化的关系，不能很好地代表声音的特征，因此，必须将声音波形转换为声学特征向量。有许多声音特征提取方法，如梅尔频率倒谱系数（MFCC）、线性预测倒谱系数（LPCC）、多分辨率耳蜗图（Multi-Resolution CochleaGram，MRCG）。

（2）分类器

我们可以使用以下 4 种类型的基于 MRCG 的分类器。这些分类器由附带 TensorFlow 的 Python 实现。

- 自适应上下文关注模型（ACAM）。
- 增强的深度神经网络（bDNN）。
- 深度神经网络（DNN）。
- 长短期记忆递归神经网络（LSTM-RNN）。

4.22　使用 numpy

使用 numpy 可以加载.npy 格式的数据，然后使用 matplotlib 显示图像。实现代码如下：

```
import numpy as np;
c = np.load( "F:/book/models-master/official/mnist/examples.npy" );
import matplotlib.pyplot as plt
plt.imshow(c[0], cmap=plt.cm.gray)
plt.show()
```

第 5 章 Java 开发语音识别

Java 中的 javac 命令从命令提示符窗口编译程序——它从文本文件中读取 Java 源程序并创建编译的 Java 类文件。javac 命令的基本语法格式为：

```
javac filename [options]
```

例如，要编译名为 HelloWorld.java 的程序，可以使用以下命令：

```
javac HelloWorld.java
```

执行上述命令，会将 HelloWorld 引用的类一起编译。

javac 命令既可以实现编译用户在命令行中指定的单个文件，也可以使用以下任意方法实现一次编译多个文件。

方法 1：在 javac 命令中列出要编译的多个文件名。例如，使用以下命令同时编译 3 个文件：

```
javac TestProgram1.java TestProgram2.java TestProgram3.java
```

方法 2：使用通配符编译一个文件夹中的所有文件。例如：

```
javac *.java
```

方法 3：如果需要同时编译大量文件（可能不是文件夹中的所有文件），又不想使用通配符，则可以先创建参数文件，列出要编译的文件。在参数文件中，可以根据需要输入文件名，文件名之间使用空格或换行符分隔。例如，这里有一个名为 TestPrograms 的参数文件，列出了 3 个要编译的文件：

```
    TestProgram1.java
```

```
    TestProgram2.java
    TestProgram3.java
```

然后，在 javac 命令行中使用“@+参数文件名”的形式编译该文件中所有的程序。本示例实现命令如下：

```
javac @TestPrograms
```

5.1 实现卷积

卷积是语音信号处理的基础。下面为二维卷积的实现代码。

实现单点卷积：

```
public static int singlePixelConvolution(int[][] input, //输入二维矩阵
        int x, int y,                   //计算单点卷积的位置
        int[][] k,                      //卷积核
        int kernelWidth,                //卷积核宽度
        int kernelHeight) {             //卷积核高度
    int output = 0;
    for (int i = 0; i < kernelWidth; ++i) {
        for (int j = 0; j < kernelHeight; ++j) {
            output = output + (input[x + i][y + j] * k[i][j]);
        }
    }
    return output;
}
```

根据单点卷积实现二维卷积：

```
public static int[][] convolution2D(int[][] input, //输入二维矩阵
        int width, int height,          //计算卷积的宽度和高度
        int[][] kernel,                 //卷积核
        int kernelWidth,                //卷积核宽度
        int kernelHeight) {             //卷积核高度
    int smallWidth = width - kernelWidth + 1;          //计算输出宽度
    int smallHeight = height - kernelHeight + 1;       //计算输出高度
    int[][] output = new int[smallWidth][smallHeight]; //输出特征图
```

```
        for (int i = 0; i < smallWidth; ++i) {
            for (int j = 0; j < smallHeight; ++j) {
                output[i][j] = 0;
            }
        }
        for (int i = 0; i < smallWidth; ++i) {
            for (int j = 0; j < smallHeight; ++j) {
                output[i][j] = singlePixelConvolution(input, i, j, kernel, kernelWidth,
kernelHeight);
            }
        }
        return output;
    }
```

测试二维卷积：

```
    public static void main(String[] args) {
        int[][] image = { { 1, 2, 3, 4, 5, 6, 7 },
                { 8, 9, 10, 11, 12, 13, 14 },
                { 15, 16, 17, 18, 19, 20, 21 },
                { 22, 23, 24, 25, 26, 27, 28 },
                { 29, 30, 31, 32, 33, 34, 35 },
                { 36, 37, 38, 39, 40, 41, 42 },
                { 43, 44, 45, 46, 47, 48, 49 } };
        int[][] filter kernel = { { 0, -6, 2 }, { 1, 3, -2 }, { -1, 1, -1 } };
        int[][] output = convolution2D(image, image[0].length, image.length,
                              filter kernel, filter kernel[0].length,
                filter_kernel.length);
        System.out.println(Arrays.deepToString(output));
    }
```

输出结果：

```
    [[-7, -10, -13, -16, -19], [-28, -31, -34, -37, -40], [-49, -52, -55, -58,
-61], [-70, -73, -76, -79, -82], [-91, -94, -97, -100, -103]]
```

5.2 KaldiJava

用户既可以使用 Ultraedit 远程编辑服务器上的 Bash 脚本文件，然后在终端直接运

行，也可以远程修改服务器上的 KaldiJava 项目中的 Java 代码，然后在终端用 Ant、Maven 或 Gradle 编译。

5.2.1 使用 Ant

在 CentOS 下，安装 Ant。

```
#yum -y install ant
```

项目的源代码根路径包括一个 build.xml 文件，每一个 build.xml 文件只有一个 Project（工程），里面定义了这个工程的全局属性。执行 ant 时，可以选择执行哪个目标。当没有指定目标时，执行 Project 的默认属性所确定的目标，只需要执行如下命令。

```
#ant
```

build.xml 文件定义了一个项目。与项目相关的信息包括项目名和默认编译的目标。例如，项目 Kaldi 默认编译的目标为 makeJAR。

```
<project name="kaldi" default="makeJAR" basedir=".">
```

可以运行指定的任务。例如，运行下面的 compile 任务。

```
<target name="compile" depends="init"
  description="compile the source ">
  <javac compiler="modern" encoding="utf-8" debug="true" srcdir="${src}"
destdir="${bin}" classpathref="project.class.path" target="1.8" source="1.8" />
</target>
```

使用以下命令行：

```
#ant compile
```

通过 Ant 执行的 build.xml 来自动生成可执行的 jar 包。Ant 通过调用目标树，就可以执行各种目标。例如，编译源代码的目标，还有打 jar 包的目标。

由于 Ant 构建文件是 XML 格式的文件，因此很容易维护和书写，而且结构很清晰。Ant 可以集成到开发环境中，Eclipse 默认安装了 Ant 插件。选中 build.xml 后，在 run as 中选取 ant build，即可运行 build.xml 中的默认目标。

使用 build.xml 可以完成的事情如下。

- 定义全局变量，如定义项目名。
- 初始化，主要是创建目录，如发布路径。
- 编译 java 源代码成为 class 文件，调用<javac encoding="utf-8" debug="true" srcdir="${src}" destdir="${bin}" classpathref="project.class.path" target="1.8" source="1.8"/>。
- 把 class 文件打包到一个 jar 文件，调用<jar destfile="***">。
- 创建 API 文档。

目标之间可以有依赖关系，例如 makeJAR 依赖 init 和 compile，init 依赖 clean。所以目标执行顺序是 clean → init → compile → makeJAR。

```
<target name="makeJAR" depends="init,compile">
```

如果需要更新 war 中的文件，则设置 update="true"。jar 包中要正好包含有用的 class 文件，既不能包含测试部分代码，也不能包含源文件。例如：

```
<target name="makeJAR" depends="init,compile">
   <jar destfile="${dist}/${jarfile}">
    <fileset dir="${bin}">
       <include name="**/*.class"/>
      <exclude name="**/*.jflex"/>
      </fileset>
  </jar>
</target>
```

javac 标签调用 java 编译器。如果 Java 源代码文件编码不一致可能会出错，则可以把编码统一成 GBK 或 UTF-8。如果源代码文件编码为 UTF-8，则使用 javac 编译时要增加 encoding 选项并指定编码为 UTF-8。

```
<javac encoding="utf-8" debug="true" srcdir="${src}" destdir="${bin}"
classpathref="project.class.path" target="1.6" source="1.6"/>
```

在 junit 任务中可以使用<batchtest>。例如：

```
<target name="test" depends="compileTest">
```

```
    <junit>
      <classpath>
        <pathelement location="bin" />
        <pathelement location="lib/junit-4.10.jar"/>
      </classpath>
      <batchtest>
         <fileset dir="${test}">
             <include name="**/*Test*" />
         </fileset>
      </batchtest>
      <formatter type="brief" usefile="false"/>
    </junit>
  </target>
```

5.2.2　使用 Maven

使用 Maven 可以构建 Java 项目。Maven 使用 pom.xml 文件配置和管理项目。在项目的根目录中放置 pom.xml，在 src/main/java 目录中放置项目的运行代码，在 src/test/java 中放置项目的测试代码。

使用 maven archetype 来创建项目的结构。采用 Maven 构建的项目一般包括一个 pom.xml 文件。

```
  <build>
      <plugins>
          <plugin>
              <artifactId>maven-assembly-plugin</artifactId>
              <configuration>
                  <archive>
                      <manifest>
                          <mainClass>fully.qualified.MainClass</mainClass>
                      </manifest>
                  </archive>
                  <descriptorRefs>
                      <descriptorRef>jar-with-dependencies</descriptorRef>
                  </descriptorRefs>
              </configuration>
          </plugin>
```

```
    </plugins>
</build>
```

使用下面的命令执行它。

```
mvn assembly:single
```

用 install 参数下载依赖的 jar 文件。

```
mvn install
```

Maven 默认的本地仓库地址为${user.home}/.m2/repository。例如，如果用 Administrator 账户登录，则把 jar 包下载到 C:\Users\Administrator\.m2\repository\这样的路径。如果 jar 包位于 lib 路径下，则 Eclipse 的.classpath 文件中的 classpathentry 为 lib 类型。

```
<classpathentry kind="lib" path="lib/commons-io-1.2.jar"/>
```

如果 jar 包位于 Maven 的存储库中，则 Eclipse 的.classpath 文件中的 classpathentry 为 var 类型。

```
<classpathentry kind="var" path="M2_REPO/junit/junit/4.8.2/junit-4.8.2.jar"
             sourcepath="M2_REPO/junit/junit/4.8.2/junit-4.8.2-sources.jar"/>
```

5.2.3 使用 Gradle

相比 Maven，Gradle 提供了更加灵活的构建方式。可以下载二进制文件来安装 Gradle，实现代码如下：

```
#cd /opt/
#wget -bc https://services.gradle.org/distributions/gradle-3.5-bin.zip
#unzip gradle-3.5-bin.zip
#mv gradle-3.5 ./gradle
```

可以在/etc/profile 文件或/etc/profile.d 下设置环境变量。不过/etc/profile.d/比/etc/profile 好维护，对于不需要软件的变量，可以直接删除/etc/profile.d/下对应的 shell 脚本。创建设置环境变量的脚本文件：

```
#echo 'export GRADLE_HOME=/opt/gradle' > /etc/profile.d/gradle.sh
#echo 'export PATH=$PATH:$GRADLE_HOME/bin'  >> /etc/profile.d/gradle.sh
```

执行这个脚本文件：

```
#. /etc/profile.d/gradle.sh
```

检查 gradle 的环境变量是否设置正确：

```
#gradle -v
```

这里，只需要使用 gradle 编译代码。例如：

```
#gradle build
```

如果需要从本地目录中获取 jar 文件，则在 build.gradle 文件中添加以下代码。

```
repositories {
   flatDir {
      dirs 'libs'
   }
}
dependencies {
   compile name: 'commons-exec-1.3'
}
```

要创建一个 Java 项目，先创建一个新的项目目录，进入并执行：

```
#gradle init --type java-library
```

会得到源文件夹和 Gradle 构建文件。

如果使用默认 gradle 包装结构建立项目，即：

```
src/main/java
src/main/resources
src/test/java
src/test/resources
```

则不需要修改 sourceSets 来运行测试，Gradle 会发现测试类和资源都在 src/test 下。

运行一个测试用例：

```
#gradle test -Dtests.class=LibraryTest
#gradle test "-Dtests.class=*.ClassName"
```

运行包和子包中所有的测试用例：

```
#gradle test "-Dtests.class= com.lietu.package.*"
```

大多数工具需要在计算机上进行安装才能使用它们。同样重要的是，用户是否会为构建工具安装正确的版本。如果正在使用旧版本的构建软件怎么办呢？

Gradle Wrapper（以下称为 Wrapper）解决了以上两个问题，它是开始 Gradle 构建的首选方式。使用 Wrapper 可以使项目组成员不必预先安装好 Gradle，便于统一项目所使用的 Gradle 版本。

在 Windows 操作系统下可以添加环境变量 GRADLE_USER_HOME 自定义 Gradle 缓存位置。为了安装 Gradle 的 Eclipse 插件，可以从 https://github.com/eclipse/buildship 找到安装地址。在 Eclipse 菜单的 Windows→Preferences→Gradle 下设置 Gradle User Home 路径为 F:\soft\gradle-3.5\bin。为了避免下载超时，可以在项目根目录下的 gradle.properties 文件中设置超时时间。

```
systemProp.org.gradle.internal.http.connectionTimeout=120000
systemProp.org.gradle.internal.http.socketTimeout=120000
```

5.2.4 概率分布函数

对于离散型随机变量，可以直接用直方图来描述其统计规律性。例如，图像处理中的灰度图可以用直方图统计每种颜色出现的概率。一般把语音信号当作连续型随机变量。对于连续型随机变量，因为无法一一列举出随机变量的所有可能取值，所以不能像随机变量那样描述它的概率分布，于是引入概率密度函数（probability Density Function，PDF），用概率密度函数的积分来求随机变量落入某个区间的概率。概率密度函数可以被看成是直方图的平滑近似。java 实现代码如下：

```
//return phi(x) = 高斯概率密度函数
public static double phi(double x) {
   return Math.exp(-x*x/2)/Math.sqrt(2*Math.PI);
}

//return phi(x, mu, signma) = 高斯概率密度函数,均值是 mu,标准差是 sigma
public static double phi(double x, double mu, double sigma) {
```

```
        return phi((x - mu)/sigma)/sigma;
    }
```

多变量的高斯密度函数，实现代码如下：

```
    public final class GaussianDistribution{
        private double m expCoef = -1;
        private double[] m sigmaInverse;
        private int[] m featColumns;
        private double[] m means;
        public GaussianDistribution(int[] columns, double[] means, double[]
sigmas) {
            assert(means.length == sigmas.length);
            assert(columns.length == means.length);
            m featColumns = columns;
            m sigmaInverse = new double[sigmas.length];
            double det = 1.0;
            for(int i = 0; i < columns.length; ++i) {
             det*=sigmas[i];
             m_sigmaInverse[i] = 1.0/sigmas[i];
            }
            m_expCoef = Math.pow(2.0*Math.PI,-columns.length/2.0)*Math.pow(det,
-0.5);
            m means = means;
        }
        public double getPDF(float[] data) {
            double exponent = 0;
            for(int i = 0; i < m sigmaInverse.length; ++i) {
             double deviation = data[m featColumns[i]]-m means[i];
             exponent += m_sigmaInverse[i]*deviation*deviation;
            }
            exponent *=-0.5;
            return m expCoef*Math.exp(exponent);
        }
        public int getDimensions() {
            return m featColumns.length;
        }
    }
```

高斯混合模型，实现代码如下：

```
public final class GaussianMixtureModel {
    private List<GaussianDistribution> m_distributions = null;
    private double[] m_mixtureProbabilities;
    public GaussianMixtureModel(List<GaussianDistribution> dists, double[]
mixtureProbabilities) {
        m_distributions = dists;
        m_mixtureProbabilities = mixtureProbabilities;
    }
    public double getPDF(float[] data) {
        double prob = 0;
        for(int i = 0; i < m_distributions.size(); ++i) {
         prob += m_mixtureProbabilities[i]*m_distributions.get(i).getPDF(data);
        }
        return prob;
    }
}
```

5.3 TensorFlow 的 Java 接口

本节先讲解如何在 Windows 下使用 TensorFlow 的 Java 接口，然后讲解使用编译工具 Bazel 从源代码编译和使用 TensorFlow 的 Java 接口。

5.3.1 在 Windows 操作系统下使用 TensorFlow

为了使用 TensorFlow 的 Java API，可以在项目的 pom.xml 中加入以下内容：

```
<dependency>
   <groupId>org.tensorflow</groupId>
   <artifactId>tensorflow</artifactId>
   <version>1.9.0-rc2</version>
</dependency>
```

增加对 JUnit 5 单元测试的支持：

```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-engine</artifactId>
   <version>${junit.jupiter.version}</version>
   <scope>test</scope>
</dependency>
```

这里的“${junit.jupiter.version}”是 Maven 属性。

TensorFlow 软件使用 Tensor 来表示计算中的数据。Tensor 类是一个带类型的多维数组。例如，一个浮点数标量：

```
Tensor tx = Tensor.create(1.0f);
```

一个一维向量：

```
float[] x = new float[]{1,2,3,4,5,6,7,8,9,0,1,2,30};
Tensor tx = Tensor.create(x);
System.out.println(tx);  //输出 Tensor 的形状
```

使用 org.junit.jupiter.api.Assertions.assertEquals 方法判断 Tensor 的形状：

```
float[] x = new float[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 30 };
Tensor tx = Tensor.create(x);
assertEquals(1, tx.shape().length, "length should equal 1");
```

一个 3×2 的浮点数矩阵：

```
float[][] matrix = new float[3][2];
Tensor m = Tensor.create(matrix);
System.out.println(m.numDimensions()); //输出维度
```

创建一个常量：

```
Graph g = new Graph(); //创建计算图
Tensor c1 = Tensor.create(3.0f);
g.opBuilder("Const", "MyConst").setAttr("dtype", c1.dataType()).setAttr
("value", c1).build();
```

两个常量相加：

```
Graph g = new Graph();
//创建会话
Session s = new Session(g);
```

```
    Tensor node1 = Tensor.create(1.0f);
    Tensor node2 = Tensor.create(4.0f);
    System.out.println(node1.floatValue());
    System.out.println(node2.floatValue());
    //往计算图添加两个常量
    Operation o1 =
        g.opBuilder("Const", "node1").setAttr("dtype", node1.dataType())
         .setAttr("value", node1).build();
    Operation o2 =
        g.opBuilder("Const", "node2").setAttr("dtype", node2.dataType())
          .setAttr("value", node2).build();
    Operation node3 =
        g.opBuilder("Add","sum").addInput(o1.output(0)).addInput(o2.output(0)).
build();
    System.out.println(node3);
    List<Tensor> results = s.runner().fetch("sum").run(); //运行计算图,并获取结果
    System.out.println(results.get(0).floatValue());
    //关闭会话
    s.close();
```

5.3.2 在 Linux 操作系统下使用 TensorFlow

本小节讲解在 Linux 操作系统下使用 TensorFlow 的 Java 接口。

安装 JDK 8：

```
    #sudo apt-get install openjdk-8-jdk
```

添加 Bazel 分发 URI 作为包源：

```
    #echo  "deb  [arch=amd64]  http://storage.googleapis.com/bazel-apt  stable
jdk1.8" | sudo tee /etc/apt/sources.list.d/bazel.list
    curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key add -
```

安装和更新 Bazel：

```
    #sudo apt-get update && sudo apt-get install bazel
```

安装后，可以使用以下命令升级到 Bazel 的较新版本。

```
    #sudo apt-get upgrade bazel
```

接下来通过编译一个简单的 C++程序来测试 Bazel。在这里，使用 Bazel 提供的 cc_binary 规则构建 C++二进制文件。在 cc_binary 规则中，二进制文件的名称在 name 属性中指定（在本示例中为 hello-world），要构建的必需源文件在 srcs 属性中指定。

```
cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
)
```

hello-world.cc 中的源代码内容如下：

```
#include <ctime>
#include <string>
#include <iostream>
std::string get greet(const std::string& who) {
   return "Hello " + who;
}

void print localtime() {
   std::time t result = std::time(nullptr);
   std::cout << std::asctime(std::localtime(&result));
}

int main(int argc, char** argv) {
   std::string who = "world";
   if (argc > 1) {
      who = argv[1];
   }
   std::cout << get greet(who) << std::endl;
   print localtime();
   return 0;
}
```

测试 main 目录下的构建文件：

```
$bazel build --jobs 2 //main:hello-world
```

运行构建出来的可执行程序 bazel-bin/main/hello-world：

```
$./bazel-bin/main/hello-world
```

输出结果：

```
Hello world
Tue Jul 10 08:01:43 2018
```

清除现有构建：

```
$bazel clean
```

在 Linux 下编译 Java 接口：

```
$nohup bazel build  --jobs 1 --config opt   //tensorflow/java:tensorflow
      //tensorflow/java:libtensorflow_jni  &
```

编译出 libtensorflow_framework.so 和 libtensorflow_jni.so 这两个本地库文件，以及 libtensorflow.jar 这个 jar 包。

加载并测试 Java 接口：

```
File f = new File("");
System.out.println(f.getAbsolutePath());
//得到 libtensorflow_jni.so 的绝对路径
String filename = f.getAbsolutePath()+"/lib/libtensorflow jni.so";
System.load(filename);  //加载动态链接库
System.out.println("I'm using TensorFlow version: " +  TensorFlow.version());
```

第 6 章 语音信号处理

根据一段语音中的频率变化来切分语音。语音信号将是许多不同频率的混合，因此从频谱而不是单一频率考虑可能更有成果。对于固定音高的持续音符，除了音符的基本频率之外，还会出现大量的泛音和谐波。而对于实际的语音，由于元音和辅音的不同音调特性，即使在短片段内频谱也会剧烈变化。

6.1 使用 FFmpeg

使用 FFmpeg 可以实现各种音频格式间的转换或从视频中提取音频。

在 CentOS 7 下，安装 YUM 源：

```
sudo rpm --import http://li.nux.ro/download/nux/RPM-GPG-KEY-nux.ro
sudo rpm -Uvh http://li.nux.ro/download/nux/dextop/el7/x86_64/nux-dextop-release-0-5.el7.nux.noarch.rpm
```

安装 FFmpeg 和 FFmpeg 开发包：

```
sudo yum install ffmpeg ffmpeg-devel -y
```

测试是否安装成功：

```
#ffmpeg
```

如果想了解更多有关 FFmpeg 的命令行参数，可以输入：

```
#ffmpeg -h
```

使用 FFmpeg 将.wav 格式转换为.mp3 格式：

```
#ffmpeg -i A.wav -acodec libmp3lame A.mp3
```

将.ogg 格式转换为.mp3 格式：

```
#ffmpeg -i audio.ogg -acodec libmp3lame audio.mp3
```

将.ac3 格式转换为.mp3 格式：

```
#ffmpeg -i audio.ac3 -acodec libmp3lame audio.mp3
```

将.aac 格式转换为.mp3 格式：

```
#ffmpeg -i audio.aac -acodec libmp3lame audio.mp3
```

6.2 标注语音

标注语音的字母表可以使用 IPA 或 SAMPA。SAMPA 是计算机可读的语音字母表，它基本上包括将国际音标的符号映射到 33～127 范围内的 ASCII 码，即 7 位可打印的 ASCII 字符。SAMPA 和 IPA 的对照表如表 6-1 所示。

表 6-1　SAMPA 和 IPA 的对照表

	SAMPA	IPA	说　明	例　子
元音	A	ɑ	开前不圆唇元音	英语的 start
	{	æ	次开前不圆唇元音	英语的 map
	6	ɐ	次开央元音	德语的 besser
	Q	ɒ	开后圆唇元音	英语的 lot
	E	ɛ	半开前不圆唇元音	英语的 met
	@	ə	中央元音	英语的 banana
	3	ɜ	半开央不圆唇元音	英语的 nurse
	I	ɪ	次闭次前不圆唇元音	英语的 kit
	O	ɔ	半开后圆唇元音	英语的 thought

续表

	SAMPA	IPA	说 明	例 子
元音	2	ø	半闭前圆唇元音	法语的 deux
	9	œ	半开前圆唇元音	法语的 neuf
	&	ɶ	开前圆唇元音	瑞典语的 skörd
	U	ʊ	次闭次后圆唇元音	英语的 foot
	}	ʉ	闭央圆唇元音	瑞典语的 sju
	V	ʌ	半开后不圆唇元音	英语的 strut
	Y	ʏ	次闭次前圆唇元音	德语的 hübsch
辅音	B	β	浊双唇擦音	西班牙语的 cabo
	C	ç	清硬颚擦音	德语的 ich
	D	ð	浊齿擦音	英语的 then
	G	ɣ	浊软颚擦音	西班牙语的 fuego
	L	ʎ	硬腭边近音	英语的 million
	J	ɲ	硬颚鼻音	英语的 canyon
	N	ŋ	软腭鼻音	英语的 thing
	R	ʁ	浊小舌擦音	法语的 roi
	S	ʃ	清龈后擦音	英语的 ship
	T	θ	清齿擦音	英语的 thin
	H	ɥ	唇硬颚近音	法语的 huit
	Z	ʒ	浊后龈擦音	英语的 measure
	?	ʔ	喉塞音	丹麦语的 stød

使用 Audacity 可以标注音轨。Audacity 标签轨的基本格式为：

```
开始时间 结束时间 文本标签
```

例如：

```
3.721154 4.045673 E
```

使用开源的 PRAAT（http://www.praat.org/）可以标注出音节边界。

6.3 时间序列

支持向量的时间序列点：

```
public class TimeSeriesPoint {
    private final double[] measurements;  //向量值
    private final int hashCode;
    public TimeSeriesPoint(double[] values) {
        int hashCode = 0;
        measurements = new double[values.length];
        for (int i = 0; i < values.length; i++) {
            hashCode += new Double(values[i]).hashCode();
            measurements[i] = values[i];
        }
        this.hashCode = hashCode;
    }

    public double get(int dimension) {
        return measurements[dimension];
    }

    public double[] toArray() {
        return measurements;
    }

    public int size() {
        return measurements.length;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        else if (o instanceof TimeSeriesPoint) {
            final double[] testValues = ((TimeSeriesPoint) o).toArray();
            if (testValues.length == measurements.length) {
                for (int x = 0; x < measurements.length; x++)
                    if (measurements[x] != testValues[x])
```

```
                    return false;
                return true;
            } else
                return false;
        } else
            return false;
    }

    @Override
    public int hashCode() {
        return hashCode;
    }
}
```

时间序列项：

```
public final class TimeSeriesItem {
    private final double time;
    private final TimeSeriesPoint point;
    public TimeSeriesItem(double time, TimeSeriesPoint point) {
        this.time = time;
        this.point = point;
    }

    public double getTime() {
        return time;
    }

    public TimeSeriesPoint getPoint() {
        return point;
    }
}
```

6.4　端点检测

语音信号处理中的端点检测主要是为了自动检测出语音的起始点及结束点。这里

我们采用了双门限比较法来进行端点检测。双门限比较法以短时能量 E 和短时平均过零率 Z 作为特征，结合两者的优点，使检测更为准确，且能够有效降低系统的处理时间、排除无声段的噪声干扰，从而提高语音信号的处理性能。

```
DataBlocker b = new DataBlocker(10); //means 10ms
SpeechClassifier s = new SpeechClassifier(10, 0.003, 10, 0);
b.setPredecessor(dataSource);
s.setPredecessor(b);
Data d = s.getData();
while(d != null) {
    if(s.isSpeech()) {
        System.out.println("Speech is detected");
    }
    else {
        System.out.println("Speech has not been detected");
    }

    System.out.println();
    d = s.getData();
}
```

6.5 动态时间规整

对于时间序列，以经常使用的欧氏距离来计算相似度存在着其很明显的缺陷。举个比较简单的例子，假设有序列 ts1 为 1,1,1,10,2,3，序列 ts2 为 1,1,1,2,10,3，如果用欧氏距离，也就是 distance[i][j]=(b[j]-a[i])*(b[j]-a[i])来计算，则总的距离和应该为 128，应该说这个序列距离是非常大的。这种情况下就有人开始考虑寻找新的计算时间序列距离的方法，提出了 DTW（Dynamic Time Warping）算法，这种算法在语音识别、机器学习方面起着重要的作用。这个算法是基于动态规划的思想，解决了发音长短不一的模板匹配问题。简单来说，就是通过构建一个邻接矩阵，寻找最短路径和。

还以上面的两个序列为例，当序列 ts1 中的 10 和序列 ts2 中的 2 对应及序列 ts1 中

的 2 和序列 ts2 中的 10 对应的时候，distance[3]及 distance[4]肯定是非常大的，这就直接导致了最后距离和的膨胀。这种时候，我们需要来调整时间序列。如果让序列 ts1 中的 10 和序列 ts2 中的 10 对应，序列 ts1 中的 1 和序列 ts2 中的 2 对应，那么最后的距离和就大大缩短，这种方式可以看作是一种时间扭曲。看到这里的时候，相信会有人提出“为什么不能使序列 ts1 中的 2 与序列 ts2 中的 2 对应”的问题，那样距离和肯定是 0，距离应该是最小的，但这种情况是不允许的，因为序列 ts1 中的 10 是发生在 2 的前面，而序列 ts2 中的 2 则发生在 10 的前面，对应方式交叉会导致时间上的混乱，不符合因果关系。两个序列对齐的方式如图 6-1 所示。

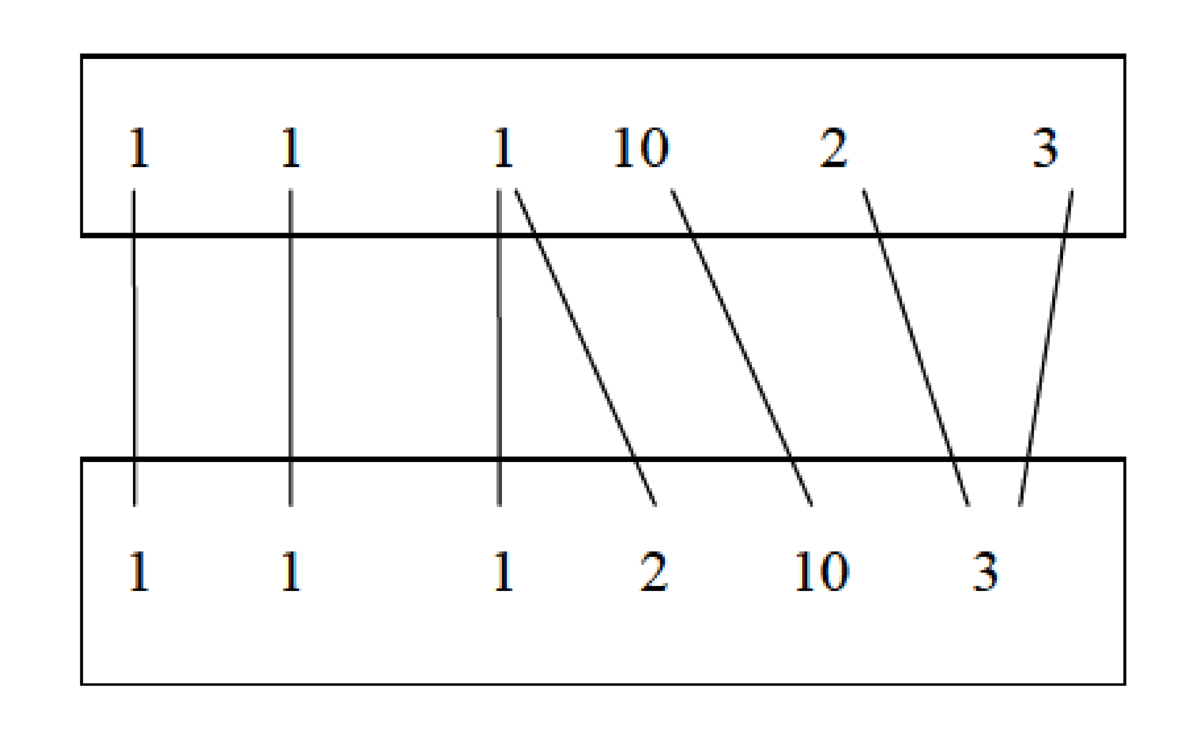


图 6-1 两个序列对齐

接下来，以 output[5][5]（所有的记录下标从 0 开始，开始时全部置 0）记录 ts1 和 ts2 之间的 DTW 距离。这个算法其实就是一个简单的动态规划，循环等式为 output[i][j]= Min(Min(output[i−1][j],output[i][j−1]),output[i−1][j−1])+ distance[i][j]，最后得到的 output[5][5]就是我们所需要的 DTW 距离。DTW 距离计算依赖关于图如图 6-2 所示。

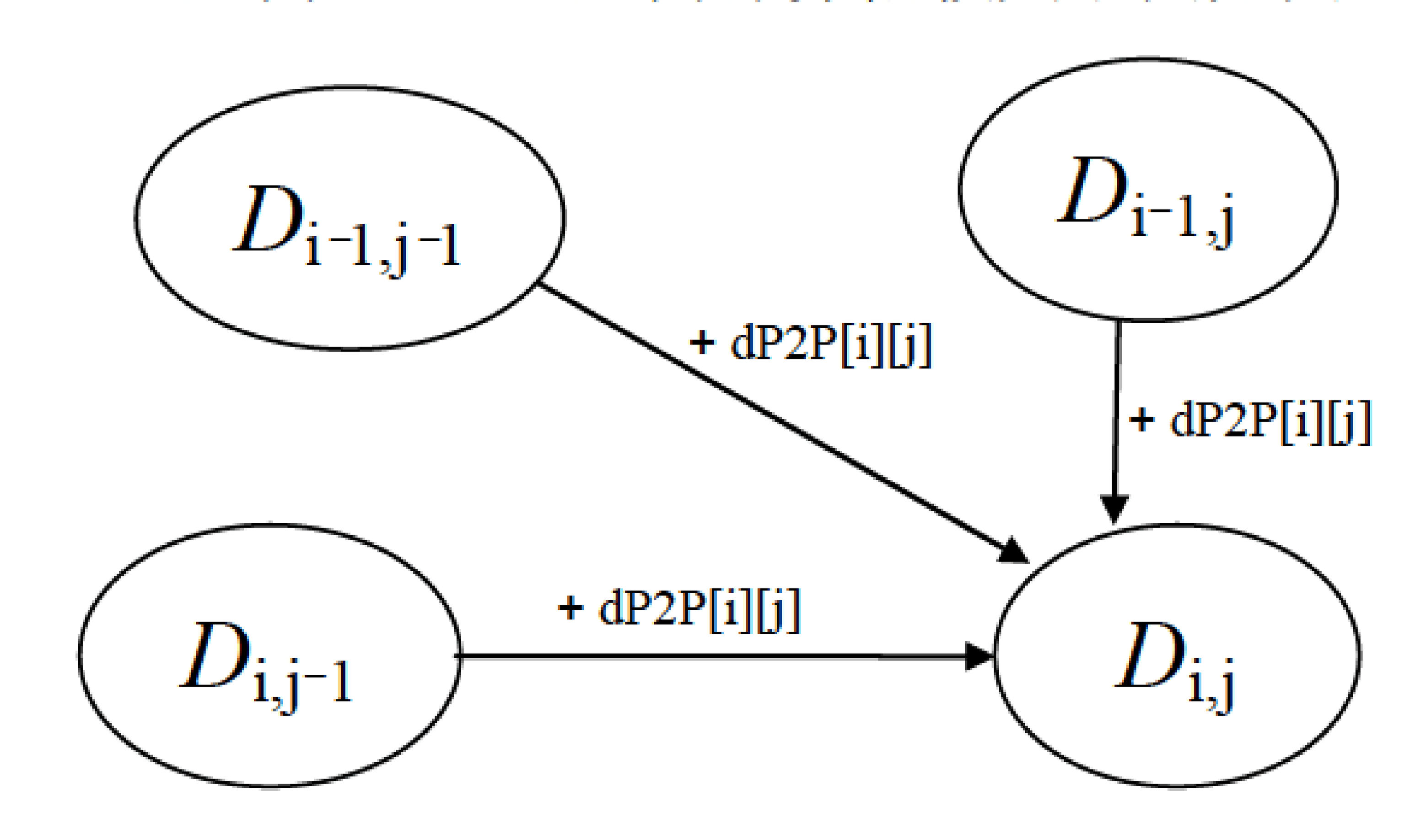


图 6-2 DTW 距离计算依赖关系图

DTW 距离实现代码如下：

```
//计算两个点的相似度
private static double pointDistance(int i, int j, double[] ts1, double[] ts2) {
      double diff = ts1[i] - ts2[j];
      return (diff * diff);
}

//计算两个序列 ts1 和 ts2 的距离
public static double DTWDistance(double[] ts1, double[] ts2) {
    int i, j;

    //构建一个点对点距离矩阵
    double[][] dP2P = new double[ts1.length][ts2.length];
    for (i = 0; i < ts1.length; i++) {
        for (j = 0; j < ts2.length; j++) {
            dP2P[i][j] = pointDistance(i, j, ts1, ts2);
        }
    }

    //用动态规划算法构建最优距离矩阵
    double[][] D = new double[ts1.length][ts2.length];
    D[0][0] = dP2P[0][0]; //开始点
    for (i = 1; i < ts1.length; i++) { //用最优值填充距离矩阵的第一列
        D[i][0] = dP2P[i][0] + D[i - 1][0];
    }
    for (j = 1; j < ts2.length; j++) { //用最优值填充距离矩阵的第一行
        D[0][j] = dP2P[0][j] + D[0][j - 1];
    }
    for (i = 1; i < ts1.length; i++) { //填充剩下的
        for (j = 1; j < ts2.length; j++) {
            double[] steps = { D[i - 1][j - 1], D[i - 1][j], D[i][j - 1] };
            double min = Math.min(steps[0], Math.min(steps[1], steps[2]));
            D[i][j] = dP2P[i][j] + min;
        }
    }
    i = ts1.length - 1;
    j = ts2.length - 1;
```

```
        return D[i][j];
    }
```

测试：

```
    public static void main(String[] args) {
        double[] ts1 = { 1,1,1,10,2,3};  //第一个序列
        double[] ts2 = { 1,1,1,2,10,3};  //第二个序列
        System.out.println(DTWDistance(ts1,ts2));  //输出两个序列的相似度
    }
```

上述代码输出结果为 2。因为序列 ts1 中的 1 对应序列 ts2 中的 2，序列 ts1 中的 2 对应序列 ts2 中的 3，这两点差异加起来为 2。

6.6　傅里叶变换

将音频数据提取到一个数组后，可以将 N 个数据样本传递给计算离散傅里叶变换的函数（或更有效地快速傅里叶变换）。

6.6.1　离散傅里叶变换

离散傅里叶变换（DFT）是数字信号处理（DSP）的一种基本但非常通用的算法。这里从头开始逐步讲解实现算法的步骤。

音频信号可以分解成为不同振幅、频率、相位的正弦波叠加。波的相位用复数来表示。离散傅里叶变换总体上是一个将 n 个复数的向量映射到 n 个复数的另一个向量的函数。使用基于 0 的索引，令 $x(t)$表示输入向量的第 t 个元素，并让 $X(k)$表示输出向量的第 k 个元素，然后基本的离散傅里叶变换由以下公式给出：

$$X(k)=\sum_{t=0}^{n-1}x(t)\mathrm{e}^{-2\pi itk/n}$$

其中，向量 $\boldsymbol{x}$ 表示各个时间点的信号水平；向量 $\boldsymbol{X}$ 表示各个频率下的信号水平。上述公式的含义是，频率 k 处的信号水平等于每个时间 t 的信号水平乘以复数指数的和。

离散傅里叶变换采用 n 个复数的输入向量，并计算 n 个复数的输出向量。由于 Java 没有原生的复数类型，所以我们将用一对实数手动模拟一个复数。向量是一个数字序列，可以用数组表示。编写一个骨架方法：

```
void dft(double[] inreal , double[] inimag,
     double[] outreal, double[] outimag) {
    //假设 4 个数组具有相同的长度
    int n = inreal.length;
    //未完成的方法体
}
```

编写外部循环，为每个输出元素分配一个值：

```
void dft(double[] inreal , double[] inimag,
         double[] outreal, double[] outimag) {
    int n = inreal.length;
    for (int k = 0; k < n; k++) {  //对于每个输出元素
        outreal[k] = ?;  //未完成
        outimag[k] = ?;  //未完成
    }
}
```

虽然看起来求和符号很吓人，但是实际上很容易理解。有限求和的一般形式仅仅意味着：

$$\sum_{j=a}^{b} f(j) = f(a) + f(a+1) + \cdots + f(b-1) + f(b)$$

下面看看我们如何取代 j 的值。在代码中，它看起来像这样：

```
double sum = 0;
for (int j = a; j <= b; j++) {
    sum += f(j);
}
//sum 的值现在有了想要的答案
```

复数的加法很容易：

$$(a+bi)+(c+di) = (a+c)+(b+d)i$$

复数的乘法稍微难一些，使用分配律和 $i^2 = -1$，得

$$(a+bi)(c+di) = ac + adi + bci - bd = (ac-bd) + (ad+bc)i$$

欧拉公式告诉我们，对于任何实数 x，有

$$e^{xi} = \cos x + i\sin x$$

而且余弦是偶函数，因此 $\cos(-x)=\cos x$；正弦是奇函数，所以 $\sin(-x)=-\sin x$。通过替换，得

$$e^{-2\pi itk/n} = e^{(-2\pi tk/n)i} = \cos\left(-2\pi\frac{tk}{n}\right) + i\sin\left(-2\pi\frac{tk}{n}\right) = \cos\left(2\pi\frac{tk}{n}\right) - i\sin\left(2\pi\frac{tk}{n}\right)$$

设 $\mathrm{Re}(x)$是 x 的实部，并设 $\mathrm{Im}(x)$是 x 的虚部。根据定义，$x = \mathrm{Re}(x) + i\mathrm{Im}(x)$，因此

$$x(t)e^{-2\pi itk/n} = [\mathrm{Re}(x(t))+i\mathrm{Im}(x(t))][\cos\left(2\pi\frac{tk}{n}\right) - i\sin\left(2\pi\frac{tk}{n}\right)]$$

展开复数乘法，得

$$x(t)\,e^{-2\pi itk/n} = [\mathrm{Re}(x(t))\cos\left(2\pi\frac{tk}{n}\right) + \mathrm{Im}(x(t))\sin\left(2\pi\frac{tk}{n}\right)]$$

$$+\,i[-\mathrm{Re}(x(t))\sin\left(2\pi\frac{tk}{n}\right) + \mathrm{Im}(x(t))\cos\left(2\pi\frac{tk}{n}\right)]$$

因此，总和中的每一项都有这个实部和虚部的代码：

```
double angle = 2 * Math.PI * t * k / n;
double real =  inreal[t] * Math.cos(angle) + inimag[t] * Math.sin(angle);
double imag = -inreal[t] * Math.sin(angle) + inimag[t] * Math.cos(angle);
```

将每个项目求和的代码合并到总的代码中，我们就完成了操作。

```
static void dft(double[] inreal , double[] inimag,
            double[] outreal, double[] outimag) {
   int n = inreal.length;

   for (int k = 0; k < n; k++) {     //对于每个输出元素
      double sumreal = 0;
      double sumimag = 0;
      for (int t = 0; t < n; t++) {  //对于每个输入元素
         double angle = 2 * Math.PI * t * k / n;
```

```
            sumreal +=  inreal[t] * Math.cos(angle) + inimag[t] * Math.sin(angle);
            sumimag += -inreal[t] * Math.sin(angle) + inimag[t] * Math.cos(angle);
        }
        outreal[k] = sumreal;
        outimag[k] = sumimag;
    }
}
```

6.6.2 快速傅里叶变换

有多种离散傅里叶变换的快速算法，最常见的是 Cooley-Tukey 算法。一般快速傅里叶变换（FFT）算法就是 Cooley-Tukey 算法。这一方法以分治法为策略，递归地将长度为 $N=N_1N_2$ 的离散傅里叶变换分解为长度为 N_1 的 N_2 个较短序列的离散傅里叶变换，以及与 $O(N)$个旋转因子的复数乘法。最简单的情况，假设 N=2，有：

$$X(k)=\sum_{t=0}^{n-1}x(t)\mathrm{e}^{-2\pi itk/n}=x(0)+x(1)\mathrm{e}^{-\pi ik}$$

当 k=0 时，$X(0)=x(0)+x(1)$。

当 k=1 时，$X(1)=x(0)-x(1)$。

计算给定复向量的离散傅里叶变换，并将结果存到向量中。向量的长度必须是 2 的乘方。使用 Cooley-Tukey 算法（按时间抽取基数 2 算法）的实现代码如下：

```
public static void transformRadix2(double[] real, double[] imag) {
    //长度变量
    int n = real.length;
    if (n != imag.length)
        throw new IllegalArgumentException("Mismatched lengths");
    int levels = 31 - Integer.numberOfLeadingZeros(n);  //Equal to floor(log2(n))
    if (1 << levels != n)
        throw new IllegalArgumentException("Length is not a power of 2");

    //构建三角函数表
    double[] cosTable = new double[n / 2];
    double[] sinTable = new double[n / 2];
    for (int i = 0; i < n / 2; i++) {
```

```
            cosTable[i] = Math.cos(2 * Math.PI * i / n);
            sinTable[i] = Math.sin(2 * Math.PI * i / n);
        }

        //位反转寻址置换
        for (int i = 0; i < n; i++) {
            int j = Integer.reverse(i) >>> (32 - levels);
            if (j > i) {
                double temp = real[i];
                real[i] = real[j];
                real[j] = temp;
                temp = imag[i];
                imag[i] = imag[j];
                imag[j] = temp;
            }
        }

        //Cooley-Tukey 以 2 为基数按时间抽取的 FFT
        for (int size = 2; size <= n; size *= 2) {
            int halfsize = size / 2;
            int tablestep = n / size;
            for (int i = 0; i < n; i += size) {
                for (int j = i, k = 0; j < i + halfsize; j++, k += tablestep) {
                    int l = j + halfsize;
                    double tpre =  real[l] * cosTable[k] + imag[l] * sinTable[k];
                    double tpim = -real[l] * sinTable[k] + imag[l] * cosTable[k];
                    real[l] = real[j] - tpre;
                    imag[l] = imag[j] - tpim;
                    real[j] += tpre;
                    imag[j] += tpim;
                }
            }
            if (size == n)  //防止溢出
                break;
        }
    }
```

计算任意长度向量的 FFT 需要使用循环卷积。计算给定复数向量的循环卷积，每

个向量的长度必须相同。

```
public static void convolve(double[] xreal, double[] ximag,
    double[] yreal, double[] yimag, double[] outreal, double[] outimag) {
    int n = xreal.length;
    if (n != ximag.length || n != yreal.length || n != yimag.length
          || n != outreal.length || n != outimag.length)
        throw new IllegalArgumentException("Mismatched lengths");
    xreal = xreal.clone();
    ximag = ximag.clone();
    yreal = yreal.clone();
    yimag = yimag.clone();
    //计算给定复向量的离散傅里叶变换,并将结果存到向量中
    transform(xreal, ximag);
    transform(yreal, yimag);

    for (int i = 0; i < n; i++) {
        double temp = xreal[i] * yreal[i] - ximag[i] * yimag[i];
        ximag[i] = ximag[i] * yreal[i] + xreal[i] * yimag[i];
        xreal[i] = temp;
    }
    inverseTransform(xreal, ximag);  //逆变换

    for (int i = 0; i < n; i++) {  //缩放(因为这个 FFT 实现省略了它)
        outreal[i] = xreal[i] / n;
        outimag[i] = ximag[i] / n;
    }
}
```

不限制向量长度的 FFT，使用 Bluestein 线性调频 *Z* 变换算法实现。

```
public static void transformBluestein(double[] real, double[] imag) {
        //找出 2 的幂卷积长度 m,使得 m>=n*2+1
        int n = real.length;
        if (n != imag.length)
            throw new IllegalArgumentException("Mismatched lengths");
        if (n >= 0x20000000)
            throw new IllegalArgumentException("Array too large");
        int m = Integer.highestOneBit(n) * 4;
```

```
        //构建三角函数表
        double[] cosTable = new double[n];
        double[] sinTable = new double[n];
        for (int i = 0; i < n; i++) {
          int j = (int)((long)i * i%(n * 2));  //这比 j=i*i 更准确
          cosTable[i] = Math.cos(Math.PI * j / n);
          sinTable[i] = Math.sin(Math.PI * j / n);
        }

        //临时向量和预处理
        double[] areal = new double[m];
        double[] aimag = new double[m];
        for (int i = 0; i < n; i++) {
          areal[i] =  real[i] * cosTable[i] + imag[i] * sinTable[i];
          aimag[i] = -real[i] * sinTable[i] + imag[i] * cosTable[i];
        }
        double[] breal = new double[m];
        double[] bimag = new double[m];
        breal[0] = cosTable[0];
        bimag[0] = sinTable[0];
        for (int i = 1; i < n; i++) {
          breal[i] = breal[m - i] = cosTable[i];
          bimag[i] = bimag[m - i] = sinTable[i];
        }

        //卷积
        double[] creal = new double[m];
        double[] cimag = new double[m];
        convolve(areal, aimag, breal, bimag, creal, cimag);

        //后处理
        for (int i = 0; i < n; i++) {
          real[i] =  creal[i] * cosTable[i] + cimag[i] * sinTable[i];
          imag[i] = -creal[i] * sinTable[i] + cimag[i] * cosTable[i];
        }
    }
```

计算给定复向量的离散傅里叶变换，并将结果存到向量中，向量可以有任何长度。

```
public static void transform(double[] real, double[] imag) {
      int n = real.length;
      if (n != imag.length)
        throw new IllegalArgumentException("Mismatched lengths");
      if (n == 0)
        return;
      else if ((n & (n - 1)) == 0)  //是 2 的幂
        transformRadix2(real, imag);
      else  //用于任意长度的更复杂算法
        transformBluestein(real, imag);
}
```

根据 FFT 的输出结果可以得到幅度和相位。幅度被编码为复数的模，例如 Audacity 中显示的频谱图，其中 Y 轴的值可以看成是复数的模（$\mathrm{sqrt}(x^2+y^2)$）。而相位被编码为角度（$a\tan^2(y,x)$）。正频率代表逆时针的圆周运动，负频率代表顺时针的圆周运动。

6.7 MFCC 特征

以梅尔倒普系数 MFCC 为例，对语音信号处理如下。

- 输入 16kHz 采样音频。
- 以一个 25ms 的窗口（每次移动 10ms 将输出一个向量序列）产生数值序列。
- 乘以 Windows 函数。例如，海明距离。
- 执行快速傅里叶变换。
- 在每个频率桶中记录能量，也就是计算每个频率区间的能量。
- 执行离散余弦变换（DCT），得到“倒谱”。
- 保留倒谱的前 13 个系数。

在深度学习中，可以直接读取声音的波形文件，不用 MFCC 特征。

6.8 说话者识别

不同的说话者在一个通用声学簇内具有不同的子空间。从通用簇偏移的子空间描述了样本的方向向量，其取决于说话者的语言文本。为了获得相关的仅包含说话者的特征，将这些向量分析为特征因子。分析的因子特征称为身份向量（identity vectors 即 I-vectors），i-vectors 在语音段中传达说话者特性。我们可以使用 i-vectors 来识别说话者，例如，通过计算两段语音表示的两个向量之间的余弦距离来作为这两段语音是否来源于同一个说话者的衡量指标。

6.9 解码

在 Kaldi 工具包中，没有单一的“规范”解码器或解码器必须满足的固定接口。

方面上有 SimpleDecoder 和 FasterDecoder 解码器，还有词图生成解码器。可用（参见词图生成解码器）。有命令行程序包装这些解码器，以便它们可以解码特定类型的模型（如 GMM），或者具有特定的特殊条件（如多类 fMLLR）。解码的命令行程序示例是 gmm-decode-simple、gmm-decode-faster、gmm-decode-kaldi 和 gmm-decode-faster-fmllr。我们已经避免创建一个可执行所有可能类型的解码命令行程序，因为这可能很快会变得难以修改和调试。

为了最小化解码器和声学建模代码之间的相互作用，我们创建了一个基类（DecodableInterface），可以将 DecodableInterface 对象视为声学模型和特征文件的包装器。这似乎是一个有点不自然的对象。但是，它的存在有一个很好的理由。声学模型和特征之间的相互作用可能非常复杂（考虑使用多个变换进行自适应），并且通过将特征从解码器中取出，可以大大简化解码器必须知道的内容。

解码器的基本操作是“解码 DecodableInterface 类型的这个对象”。解码接口如下：

```
class DecodableInterface {
  public:
```

```
    //返回对数似然值
    // “帧”从 0 开始. 在调用此函数之前,应该验证 IsLastFrame(frame-1)是否返回 false
    virtual BaseFloat LogLikelihood(int32 frame, int32 index) = 0;

    //如果这是最后一帧,则返回 true
    virtual bool IsLastFrame(int32 frame) const = 0;

    //调用 NumFramesReady()将返回这个可解码对象当前可用的帧数
    virtual int32 NumFramesReady() const {
       KALDI_ERR << "NumFramesReady() not implemented for this decodable type.";
     return -1;
    }

    //返回声学模型中的状态数
    //从 1 开始索引状态数,即从 1 到 NumIndices();这是为了与 OpenFst 兼容
    virtual int32 NumIndices() const = 0;

    virtual ~DecodableInterface() {}
};
```

1. SimpleDecoder

SimpleDecoder 是较简单的解码器，主要用于参考和调试更高度优化的解码器。SimpleDecoder 的构造函数使用 FST 进行解码，并使用解码束。

```
SimpleDecoder(const fst::Fst<fst::StdArc> &fst, BaseFloat beam);
```

解码话语是通过以下函数完成的。

```
void Decode(DecodableInterface &decodable);
```

构造一个 Decodable 对象并对其进行解码。

```
DecodableAmDiagGmmScaled gmm_decodable(am_gmm, trans_model, features,
                                       acoustic_scale);
decoder.Decode(gmm_decodable);
```

DecodableAmDiagGmmScaled 类型是一个非常简单的对象，给定一个 transition-id，从 trans_model（类型 TransitionModel）得到适当的 pdf-id，从这些特征（类型 Matrix

<BaseFloat>）中获取相应的行，得出来自 am_gmm（类型 AmDiagGmm）的可能性，并通过 acoustic_scale（类型 float）进行缩放。

调用 Decode 函数后，我们可以通过以下调用获取回溯。

```
bool GetBestPath(Lattice *fst_out);
```

输出格式是一个词图，但只包含一个路径。词图是有限状态转换器，其输入和输出标签是 FST 上的任何标签（通常分别为 transition-id 和单词），其权重包含声学、语言模型和转换权重。

SimpleDecoder 的工作原理如下。

此解码器在标记级别存储回溯。标记的类型为 SimpleDecoder::Token，它具有以下成员变量。

```
class Token {
   public:
   Arc arc_;
   Token *prev_;
   int32 ref_count_;
   Weight weight_;
...
```

上述代码中，Arc 类型的成员（这个成员是 fst :: StdArc 的 typedef）是原始 FST 中边的副本，只是添加了声学似然贡献，它包含输入和输出标签、权重和下一个状态（在 FST 中）; prev_成员用于追溯; ref_count_用于垃圾收集算法; Weight 是 fst::StdArc::Weight 的 typedef，实质上它只存储一个浮点值，表示到目前为止的累计成本。

SimpleDecoder 类只包含 4 个数据成员，声明如下：

```
unordered_map<StateId, Token*> cur_toks_;
unordered_map<StateId, Token*> prev_toks_;
const fst::Fst<fst::StdArc> &fst_;
BaseFloat beam_;
```

上述成员中的最后两个（即 fst_和 beam_）在解码期间是恒定的。成员 cur_toks_和 prev_toks_分别存储当前帧和前一帧的当前活动标记。函数 Decode()的中心循环如下：

```
for(int32 frame = 0; !decodable.IsLastFrame(frame-1); frame++) {
   ClearToks(prev_toks_);
   std::swap(cur_toks_, prev_toks_);
   ProcessEmitting(decodable, frame);
   ProcessNonemitting();
   PruneToks(cur_toks_, beam_);
}
```

除了 ProcessEmitting()和 ProcessNonemitting()以外，这些语句都是不言自明的。函数 ProcessEmitting()将标记从 prev_toks_传递到 cur_toks_。它只考虑发射边（即具有非零输入标签的边）。对于 prev_toks_中的每个标记（如 tok），它会查看与标记相关联的状态（在 tok-> arc_.nextstate 中），并且对于正在发出的状态中的每个边，它会创建一个带有标记的新标记。回溯到 tok 并从该边处理 arc_字段，除了相关的权重，更新以包括声学贡献。表示到此时为止的累计成本的 weight_字段将是 tok-> weight_的总和（在半环解释中，就是乘积）和最近添加的边的权重。每次尝试向 cur_toks_添加新标记时，必须确保没有与相同 FST 状态关联的现有标记。如果有，则只保留最好的。

函数 ProcessNonemitting()仅处理 cur_toks_而不处理 prev_toks_；它传递不发射的边，即带有零/ <eps>的边作为输入标签/符号。新创建的标记将指向 cur_toks_中的其他标记，边上的权重将只是 FST 的权重。ProcessNonemitting()可能必须处理 epsilons 链，它使用队列来存储需要处理的状态。

解码后，函数 GetOutput()将从最终状态下最可能的标记追溯（考虑到它的最终概率，如果 is_final == true），并在追溯序列中为每个边产生一个线性 FST。这些可能比帧数更多，因为我们为不发射的边创建了单独的标记。

2. FasterDecoder

FasterDecoder 是一种更优化的解码器。解码器 FasterDecoder 与 SimpleDecoder 具有几乎完全相同的接口。唯一重要的新配置值是“max-active”，它控制一次可以激活的最大状态数。除了强制执行最大活动状态以外，唯一的主要区别是与数据结构相关

的状态。我们用新类型 HashList<StateId, Token*>替换 std::unordered_map<StateId, Token*>类型，其中 HashList 是为某目的创建的模板类型，存储单链表结构，其元素也可通过散列表访问，并且它提供了为新列表结构释放散列表的功能，同时提供对旧列表结构的顺序访问。这样我们就可以使用散列表来访问原来在 SimpleDecoder 中的 cur_toks_，同时仍然可以以列表的形式访问原来在 SimpleDecoder 中的 prev_toks_。

主修剪步骤 FasterDecoder 在 ProcessEmitting 中进行。从概念上讲，我们在 SimpleDecoder 中的标记是 prev_toks_，在 ProcessEmitting 之前可以使用束和指定的最大活动状态数（以较小者为准）进行修剪。实现的方法是调用一个函数 GetCutoff()，返回一个权重截止值“weight_cutoff”，此截止值适用于 prev_toks_中的标记。当通过 prev_toks_（这个变量在 FasterDecoder 中不存在，但在概念上这样理解）时，我们只处理那些比 cutoff 更好的标记。

与 cutoff 相关的 FasterDecoder 中的代码比仅具有一个修剪步骤要稍微复杂一些。基本的观察是这样的——如果你以后只是忽略大部分标记，那么创造大量的标记毫无意义。因此，在 ProcessEmitting 中的情况是，我们有 weight_cutoff，但如果知道下一帧的 weight_cutoff 值 next_weight_cutoff 是多少，那不就好了吗？然后，每当我们处理具有当前帧的声学可能性的边时，如果可能性比 next_weight_cutoff 差，就可以避免创建标记。为了知道下一个权重，必须知道两件事，即必须知道下一帧上最好的标记权重及下一帧的有效束宽度。如果 max_active 约束是有限的，则有效束宽度可以与“束”不同，并且我们使用启发式，即有效束宽度在帧与帧之间不会发生很大变化。我们尝试通过传递当前最佳的标记来估计下一帧的最佳标记权重（稍后，如果在下一帧找到更好的标记，将更新此估计）。通过使用变量 adaptive_beam，我们得到下一帧上有效束宽度的粗略上界。这始终设置为较小的“束”（指定的最大束宽度），或由 max_active 确定的有效束宽度加上 beam_delta（默认值为 0.5）。当说它是一个“粗糙的上界”时，意思是它通常大于或等于下一帧的有效束宽度。创建新标记时使用的修剪值等于对下一帧最佳标记的当前估计值加上 adaptive_beam。对于有限的 beam_delta，修剪可能比单

独的 beam 和 max_active 参数更严格，尽管在 0.5 这个取值时我们不相信这种情况经常发生。

3. BiglmDecoder

BiglmDecoder 实现了使用大型语言模型进行解码。Kaldi 有两种使用大型语言模型的基本方法：一种方法是使用小 LM 生成词图，并用大 LM 重新校正该词图（参见下面的词图生成解码器，以及 Kaldi 中的词图）；另一种方法是使用 biglm 解码器，如 BiglmFasterDecoder。基本思想是用小文法创建解码图 HCLG，并用大文法和小文法之间的差异动态组合。注意，虽然我们使用“文法”一词来与标准符号兼容，但还要考虑统计语言模型。

想象一下，小文法是 G（一个 FST），大文法是 G'。基本思想是在解码时间内搜索由三重合成 $HCLG \circ G^{-} \circ G'$。构建一个按需组合的 FST，称为 F，即 $F=G^{-} \circ G'$。然后在解码时，我们按需构建 FST$HCLG \circ F$。这样做的问题是，我们总是采用通过 G 的最差得分路径，例如，不正确地采取退避边，这将使原始 FST 分数的减法不正确。

解决上述问题的方法是使用一些有关 G 和 G' 结构的知识（假设它们是 ARPA 风格的语言模型），并将它们视为无 epsilon、确定性 FST。那就是：当从具有特定输入标签的特定状态搜索边时，如果找到输入标签，就接受它（并返回该边），否则遵循 epsilon 转换，并递归地查找具有该标签的边。我们为这种类型的 FST 创建了一个特殊的接口，称为 fst::DeterministicOnDemandFst，它有一个新函数 GetArc()，可找到具有特定输入标签的边（假设不能有多个）。G 和 G'都是 fst::DeterministicOnDemandFst 类型，它们的组合也是如此。这意味着解码器不必实现通用的合成算法；相反，每当它在 HCLG 中穿过边时，只需要更新语言模型状态（F 中的状态标识符）。解码算法几乎与基线完全相同，除了状态空间（我们使用的散列索引）不仅仅是 HCLG 中的状态，而是一对（HCLG 中的状态，F 中的状态）。但是这个解码器相对于具有相同束的解码器而言，速度仍然有点儿慢（例如，在典型设置中几乎慢 1/2），没有 biglm 部分。

原因似乎是使用 biglm 解码器，更多的状态在束中（因为 HCLG 中的状态现在可能有更多的“副本”，对应于 HCLG 中具有不同语言模型状态的不同历史）。但是，对于相同的束，biglm 解码器确实提供了比用小文法产生的词图重新校正更好的准确性。我们认为，原因是更好地修剪——biglm 解码器使用更接近最佳的语言进行维特比束修剪。当然，它仍然不如通过用大文法编译的 HCLG 获得的修剪那么好，因为 biglm 解码器仅在每次跨越一个单词时更新“好的”语言模型得分。

上述一些解码器的词图生成版本有 LatticeFasterDecoder、LatticeSimpleDecoder 和 LatticeBiglmFasterDecoder。

第 7 章 深度学习

一般而言，人工神经网络具有生物学动机，意味着它们试图模仿真实神经系统的行为。就如同真正神经系统中最小的构建单元是神经元一样，人工神经网络中最小的构建单元是人工神经元。目前往往把神经元按层次组织成包括输入层和输出层在内的多层结构。除了输入层和输出层以外，神经网络还有中间层，中间层也可以称为隐藏层。隐藏层也可能称为编码器。

浅层网络隐藏层的数量较少。虽然有研究表明浅层网络可拟合任何函数，但拟合有些函数需要非常“肥胖”的层次结构，可能一层就要成千上万个神经元。而这直接导致的后果是参数的数量增加很多。深层网络能够以比浅层网络更少的参数来更好地拟合函数。

使用交叉熵目标训练前馈 DNN 声学模型。DNN 包括简单的 DNN、TDNN（时延神经网络）和 CNN（卷积神经网络）。

7.1　神经网络基础

本节以使用神经网络实现 XOR——一个简单的线性不可分问题为例，讲解神经网络的基础知识。XOR 的神经网络结构图如图 7-1 所示。

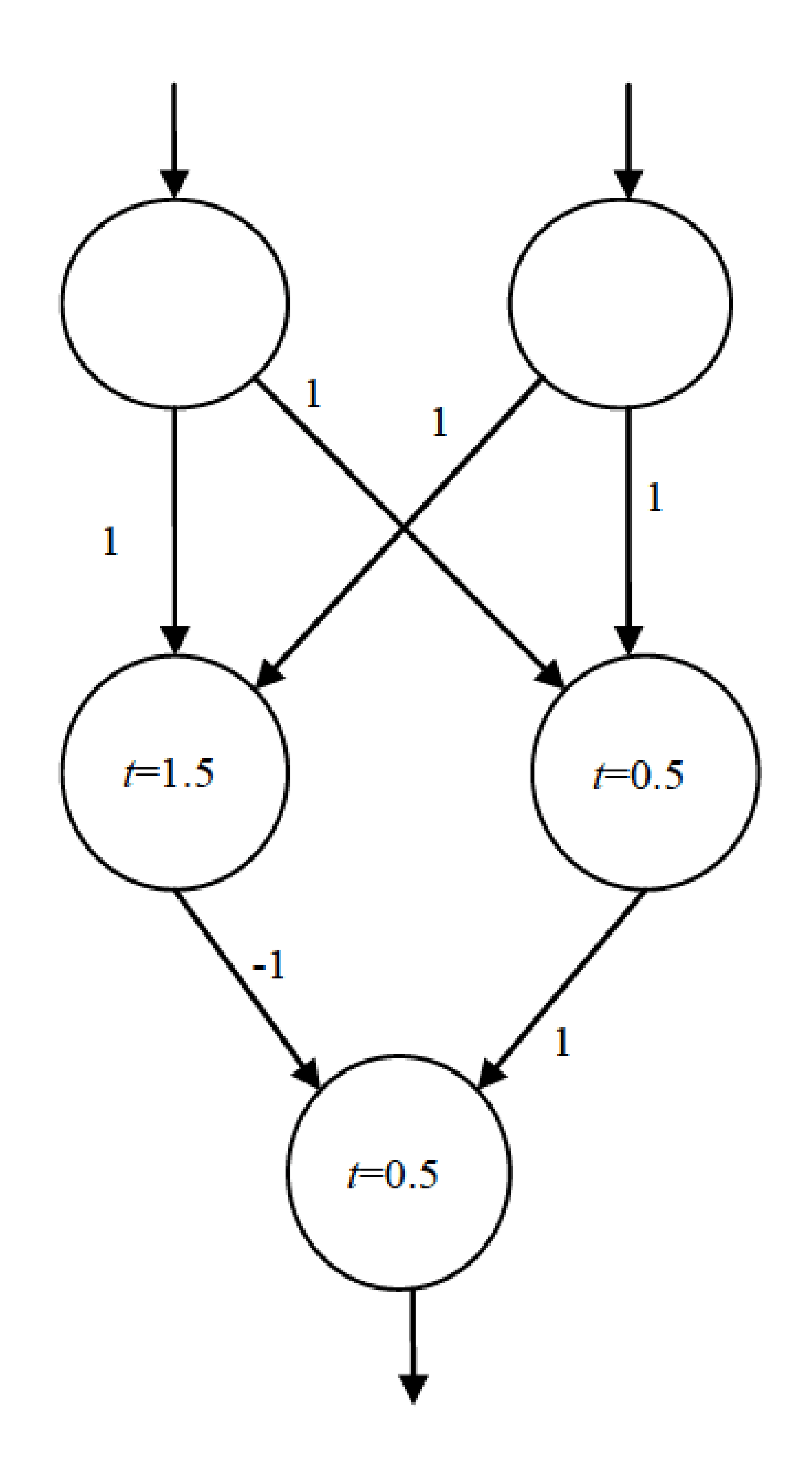


图 7-1　XOR 的神经网络结构图

神经元类实现代码如下：

```
public class Neuron {                //神经元
    private ArrayList<Neuron> inputs;
    private float weight;
    private float threshhold;        //阈值
    private boolean fired;           //是否发射
    public Neuron (float t) {
        threshhold = t;
        fired = false;
        inputs = new ArrayList<Neuron>();
    }
    public void connect (Neuron ... ns) {
        for (Neuron n : ns) inputs.add(n);
    }
    public void setWeight (float newWeight) {
        weight = newWeight;
    }
```

```
    public void setWeight (boolean newWeight) {
        weight = newWeight ? 1.0f : 0.0f;
    }
    public float getWeight () {
        return weight;
    }
    public float fire () {//发射
        if (inputs.size() > 0) {
            float totalWeight = 0.0f;
            for (Neuron n : inputs) {
                n.fire();
                totalWeight += (n.isFired()) ? n.getWeight() : 0.0f;
            }
            fired = totalWeight > threshhold;
            return totalWeight;
        }
        else if (weight != 0.0f) {
            fired = weight > threshhold;
            return weight;
        }
        else {
            return 0.0f;
        }
    }
    public boolean isFired () {
        return fired;
    }
}
```

使用这个类实现 XOR 神经网络：

```
//构建神经网络，这里是一个前馈神经网络
Neuron xor = new Neuron(0.5f);
Neuron left = new Neuron(1.5f);
Neuron right = new Neuron(0.5f);
left.setWeight(-1.0f);
right.setWeight(1.0f);
xor.connect(left, right);
```

```
for (String val : args) {
    Neuron op = new Neuron(0.0f);//创建输入神经元
    op.setWeight(Boolean.parseBoolean(val));
    //把输入神经元接入网络
    left.connect(op);
    right.connect(op);
}
xor.fire(); //触发
//输出预测结果
System.out.println("Result: " + xor.isFired());
```

输出神经元实际输出和期望之间的差距称为损失。用损失函数来衡量实际输出和期望。

7.1.1　实现多层感知器

前馈神经网络是一种简单的神经网络，各神经元分层排列。每个神经元只与前一层的神经元相连，接收前一层的输出，并输出给下一层，各层间没有反馈。

多层感知器（Multilayer Perceptron，MLP）是一种前馈人工神经网络模型，可以解决任何线性不可分问题。实现 XOR 的多层感知器网络结构如下。

第一层，即输入层，由 2 个神经元组成。

第二层，即隐藏层，有 6 个神经元。

第三层，即输出层，有 1 个神经元。

激活函数可以选择 tanh 函数()或 sigmoid 函数等。tanh()函数的范围为[-1,1]，而 sigmoid()函数的范围为[0,1]。这里，激活函数选用 tanh()函数。这个选择有以下两个原因（假设已规范化数据，这非常重要）。

- 具有更强的梯度。由于数据集中在 0 附近，这附近的导数更高。
- 避免渐变中的偏见。

选用 tanh()函数作为激活函数的神经元类实现代码修改如下：

```
public class Neuron {
   public Neuron(int prev_n_neurons, java.util.Random rand)
```

```
    {
       //每个神经元知道与前一层的每个神经元连接的权重
        synapticWeights = new float[prev n neurons];

       //设置默认权重
       for (int i = 0; i < prev n neurons; ++i)
         _synapticWeights[i] = rand.nextFloat() - 0.5f;
    }

    //用给定的输入激活神经元,然后返回输出
    public float activate(float inputs[])
    {
        activation = 0.0f;
       assert(inputs.length ==  synapticWeights.length);

       for (int i = 0; i < inputs.length; ++i) //点积
         _activation += inputs[i] * _synapticWeights[i];

       //我们的激活函数 tanh(x)
       return 2.0f / (1.0f + (float) Math.exp((-_activation) * lambda)) - 1.0f;
    }

    public float getActivationDerivative() //获得激活函数的导数
    {
       float expmlx = (float) Math.exp(lambda * _activation);
       return 2 * lambda * expmlx / ((1 + expmlx) * (1 + expmlx));
    }
    public float[] getSynapticWeights() { return  synapticWeights; }
    public float getSynapticWeight(int i) { return  synapticWeights[i]; }
    public void setSynapticWeight(int i, float v) { _synapticWeights[i] = v; }
    //--------
    private float  activation;
    private float[] _synapticWeights; //突触权重

    //sigmoid 的参数
    static final float lambda = 1.5f;
}
```

Layer 类表示感知器的一层，实现代码如下：

```
public class Layer {
   /**
    * 构造一层神经元
    * @param prev_n_neurons  前一层神经元数量
    * @param n_neurons  当前神经元数量
    * @param rand  随机对象
    */
   public Layer(int prev n neurons, int n neurons, java.util.Random rand)
   {
      //所有的层/神经元必须使用相同的随机数发生器
       n neurons = n neurons + 1;
       prev n neurons = prev n neurons + 1;

      //分配空间
       neurons = new ArrayList<Neuron>();
       outputs = new float[ n neurons];

      for (int i = 0; i <  n neurons; ++i)
         neurons.add(new Neuron( prev n neurons, rand));
   }

   //在输出向量前添加 1 个元素
   public static float[] add bias(float[] in)
   {
      float out[] = new float[in.length + 1];
      for (int i = 0; i < in.length; ++i)
        out[i + 1] = in[i];
      out[0] = 1.0f;
      return out;
   }

   //计算当前层的输出
   public float[] evaluate(float in[])
   {
      float inputs[];
      //如有必要,则添加输入(偏差)
```

```
          if (in.length != getWeights(0).length)
            inputs = add bias(in);
          else
            inputs = in;
          assert(getWeights(0).length == inputs.length);
          //刺激该层的每个神经元并获得其输出
          for (int i = 1; i <  n neurons; ++i)
             outputs[i] =  neurons.get(i).activate(inputs);

          //偏置处理
           outputs[0] = 1.0f;

          return _outputs;
       }
       public int size() { return  n neurons; }
       public float getOutput(int i) { return  outputs[i]; }
       public float getActivationDerivative(int i) {
               return  neurons.get(i).getActivationDerivative();
        }
       public float[] getWeights(int i) { return _neurons.get(i).
getSynapticWeights(); }
       public float getWeight(int i, int j) { return  neurons.get(i).
getSynapticWeight(j); }
       public void setWeight(int i, int j, float v) {  neurons.get(i).
setSynapticWeight(j, v); }

       //--------
       private int  n neurons,  prev n neurons;
       private ArrayList<Neuron>  neurons;
       private float _outputs[];
    }
```

多层感知器网络的实现代码如下：

```
    public class Mlp {
       //在构造方法中传入每层神经元数量
       public Mlp(int nn_neurons[])
       {
```

```
    Random rand = new Random();
    //创建所需的层
     layers = new ArrayList<Layer>();
    for (int i = 0; i < nn neurons.length; ++i)
      _layers.add(
              new Layer(
                      i == 0 ?
                      nn neurons[i] : nn neurons[i - 1],
                      nn neurons[i], rand)
              );

     delta w = new ArrayList<float[][]>();
    for (int i = 0; i < nn_neurons.length; ++i)
       delta w.add(new float
                  [ layers.get(i).size()]
                  [ layers.get(i).getWeights(0).length]
              );

     grad ex = new ArrayList<float[]>();
    for (int i =  0; i < nn_neurons.length; ++i)
       grad ex.add(new float[ layers.get(i).size()]);
}
public float[] evaluate(float[] inputs)
{
    //把输入传遍整个神经网络并返回输出
    assert(false);
    float outputs[] = new float[inputs.length];
    for( int i = 0; i <  layers.size(); ++i ) {
      outputs =  layers.get(i).evaluate(inputs);
      inputs = outputs;
    }
    return outputs;
}
private float evaluateError(float nn output[], float desired output[])
{
    float d[];
    //如有必要,增加偏置
```

```
        if (desired output.length != nn output.length)
            d = Layer.add bias(desired output);
        else
            d = desired output;
        assert(nn output.length == d.length);
        float e = 0;
        for (int i = 0; i < nn_output.length; ++i)
            e += (nn_output[i]-d[i])*(nn_output[i]-d[i]); //二次代价函数作为损失函数
        return e;
    }
    public float evaluateQuadraticError(ArrayList<float[]> examples,
                                        ArrayList<float[]> results)
    {
        //该函数计算给定结果集的二次型误差
        assert(false);
        float e = 0;
        for (int i = 0; i < examples.size(); ++i) {
            e += evaluateError(evaluate(examples.get(i)), results.get(i));
        }
        return e;
    }
    private void evaluateGradients(float[] results)
    {
        //遍历每层中的每个神经元
        for (int c =  layers.size()-1; c >= 0; --c) {
            for (int i = 0; i <  layers.get(c).size(); ++i) {
                //如果在处理输出层神经元
                if (c ==  layers.size()-1) {
                    grad ex.get(c)[i] =
                        2 * (_layers.get(c).getOutput(i) - results[0]) *_layers.
                        get(c).getActivationDerivative(i);
                }
                else { //如果在处理前几层的神经元
                    float sum = 0;
                    for (int k = 1; k <  layers.get(c+1).size(); ++k)
                        sum += _layers.get(c+1).getWeight(k, i) *_grad_ex.get
                        (c+1)[k];
                    _grad_ex.get(c)[i] = _layers.get(c).getActivationDerivative(i)
```

```
                * sum;
            }
        }
    }
}
private void resetWeightsDelta()
{
    //重置每个权重的增量值
    for (int c = 0; c <  layers.size(); ++c) {
        for (int i = 0; i <  layers.get(c).size(); ++i) {
            float weights[] =  layers.get(c).getWeights(i);
            for (int j = 0; j < weights.length; ++j)
                delta w.get(c)[i][j] = 0;
        }
    }
}
private void evaluateWeightsDelta()
{
    //估计每个权重的增量值
    for (int c = 1; c <  layers.size(); ++c) {
        for (int i = 0; i <  layers.get(c).size(); ++i) {
            float weights[] =  layers.get(c).getWeights(i);
            for (int j = 0; j < weights.length; ++j)
               _delta_w.get(c)[i][j] += _grad_ex.get(c)[i] * _layers.get
               (c-1).getOutput(j);
        }
    }
}
private void updateWeights(float learning rate)
{
    for (int c = 0; c <  layers.size(); ++c) {
        for (int i = 0; i <  layers.get(c).size(); ++i) {
            float weights[] =  layers.get(c).getWeights(i);
            for (int j = 0; j < weights.length; ++j)
                layers.get(c).setWeight(i, j,  layers.get(c).getWeight(i, j)
               - (learning_rate * _delta_w.get(c)[i][j]));
        }
```

```
            }
        }
        private void batchBackPropagation(ArrayList<float[]> examples,
                                          ArrayList<float[]> results,
                                          float learning rate)
        {
            resetWeightsDelta();
            for (int l = 0; l < examples.size(); ++l) {
                evaluate(examples.get(l));
                evaluateGradients(results.get(l));
                evaluateWeightsDelta();
            }
            updateWeights(learning rate);
        }
        public void learn(ArrayList<float[]> examples,
                          ArrayList<float[]> results,
                          float learning rate)
        {
            //该函数实现批量反向传播算法
            assert(false);
            float e = Float.POSITIVE INFINITY;
            while (e > 0.001f) {
                batchBackPropagation(examples, results, learning rate);
                e = evaluateQuadraticError(examples, results);
            }
        }
        private ArrayList<Layer> _layers;
        private ArrayList<float[][]> _delta_w;     //delta 值
        private ArrayList<float[]> _grad_ex;       //梯度
    }
```

使用 Mlp 训练对 XOR 分类的神经网络，实现代码如下：

```
    //初始化
    ArrayList<float[]> ex = new ArrayList<float[]>();
    ArrayList<float[]> out = new ArrayList<float[]>();
    for (int i = 0; i < 4; ++i) {
```

```
    ex.add(new float[2]);
    out.add(new float[1]);
}

//填充示例数据库
ex.get(0)[0] = -1;
ex.get(0)[1] = 1;
out.get(0)[0] = 1;
ex.get(1)[0] = 1;
ex.get(1)[1] = 1;
out.get(1)[0] = -1;
ex.get(2)[0] = 1;
ex.get(2)[1] = -1;
out.get(2)[0] = 1;
ex.get(3)[0] = -1;
ex.get(3)[1] = -1;
out.get(3)[0] = -1;

int nn_neurons[] = { ex.get(0).length,    //第一层,输入层——2 个神经元
      ex.get(0).length * 3,               //第二层,隐藏层——6 个神经元
      1 //第三层,输出层——1 个神经元
};

Mlp mlp = new Mlp(nn neurons);

for (int i = 0; i < 40000; ++i) {
    mlp.learn(ex, out, 0.3f);
    float error = mlp.evaluateQuadraticError(ex, out);
    System.out.println(i + " -> error : " + error);
}
```

7.1.2 计算过程

神经网络的基本结构如图 7-2 所示。

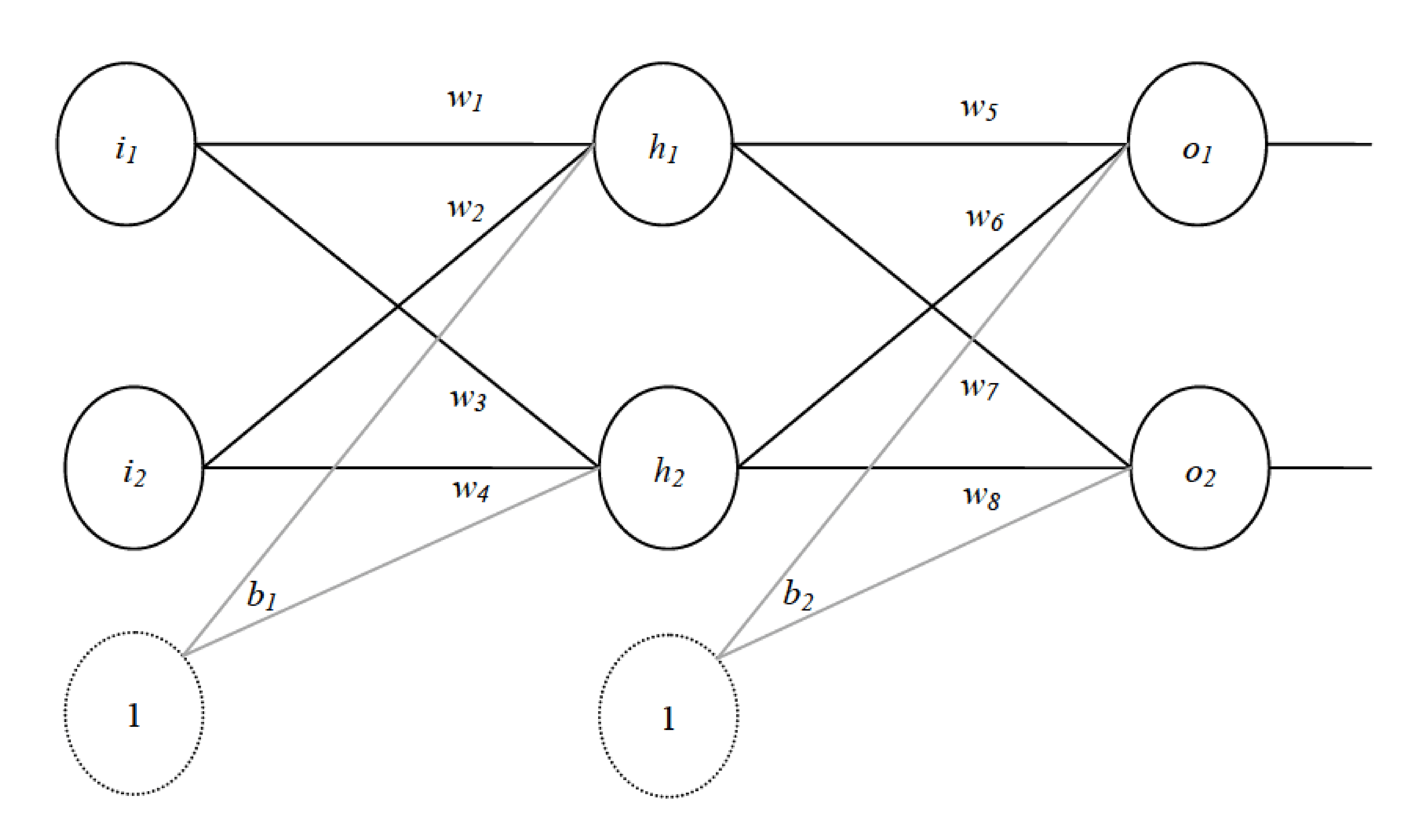


图 7-2　神经网络基本结构

使用一些数字计算，图 7-3 中是初始的权重、偏差和训练输入/输出。

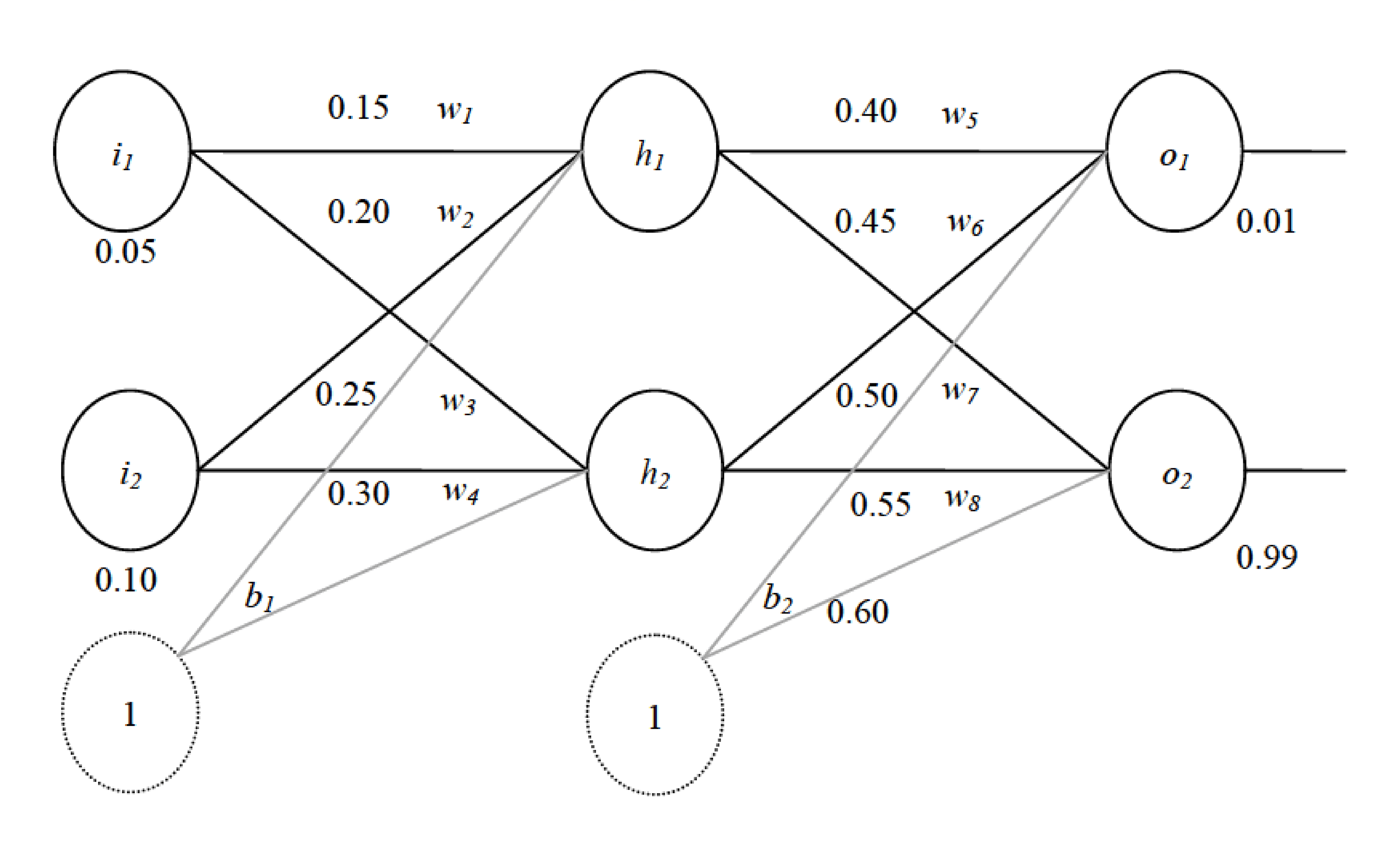


图 7-3　设置具体权值的神经网络结构

反向传播的目标是优化权重，以便神经网络可以学习如何正确映射任意输入到输出。对于其余部分，我们将使用单个训练集——给定输入 0.05 和 0.10，希望神经网络输

出 0.01 和 0.99。

首先，让我们看看神经网络在给定如图 7-3 中的权重和偏差时，对于输入 0.05 和 0.10，预测的是什么。为此，我们将通过网络向前馈传送这些输入。

计算 h_1 的总净输入：

$$net_{h_1} = w_1 \times i_1 + w_2 \times i_2 + b_1 \times 1$$

$$net_{h_1} = 0.15 \times 0.05 + 0.2 \times 0.1 + 0.35 \times 1 = 0.3775$$

使用逻辑函数对其进行压缩以获得 h_1 的输出：

$$out_{h_1} = \frac{1}{1+e^{-net_{h_1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

对 h_2 执行相同的操作，得

$$out_{h_2} = 0.596884378$$

为输出层神经元重复上述过程，使用隐藏层神经元的输出作为输入。

以下是 o_1 的输出：

$$net_{o_1} = w_5 \times out_{h_1} + w_6 \times out_{h_2} + b_2 \times 1$$

$$net_{o_1} = 0.4 \times 0.593269992 + 0.45 \times 0.596884378 + 0.6 \times 1 = 1.105905967$$

$$out_{o_1} = \frac{1}{1+e^{-net_{o_1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

对 o_2 执行相同的操作，得

$$out_{o_2} = 0.772928465$$

接下来计算总误差：

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

例如，o_1 的目标输出为 0.01，但神经网络输出为 0.75136507，因此其误差为

$$E_{o_1} = \frac{1}{2}(target_{o_1} - out_{o_1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

对 o_2 重复上述过程（目标为 0.99），得

$$E_{o_2} = 0.023560026$$

神经网络的总误差是这些误差的总和，得

$$E_{\text{total}} = E_{o_1}+E_{o_2}=0.274811083+0.023560026=0.298371109$$

我们使用反向传播的目标是更新网络中的每个权重，以便它们让实际输出更接近目标输出，从而最大限度地减少每个输出神经元和整个网络的误差。

我们想知道 w_5（某个权重值）的变化对总误差的影响有多大，也就是计算：$\dfrac{\partial E_{\text{total}}}{\partial w_5}$。

$\dfrac{\partial E_{\text{total}}}{\partial w_5}$ 读作“关于 w_5 的 E_{total} 的偏导数”，也可以说“关于 w_5 的梯度”。

通过应用求复合函数导数的链式法则，我们知道

$$\frac{\partial E_{\text{total}}}{\partial w_5} = \frac{\partial E_{\text{total}}}{\partial out_{o_1}} \times \frac{\partial out_{o_1}}{\partial net_{o_1}} \times \frac{\partial net_{o_1}}{\partial w_5}$$

用可视化的方式展现我们正在做的事情，如图 7-4 所示。

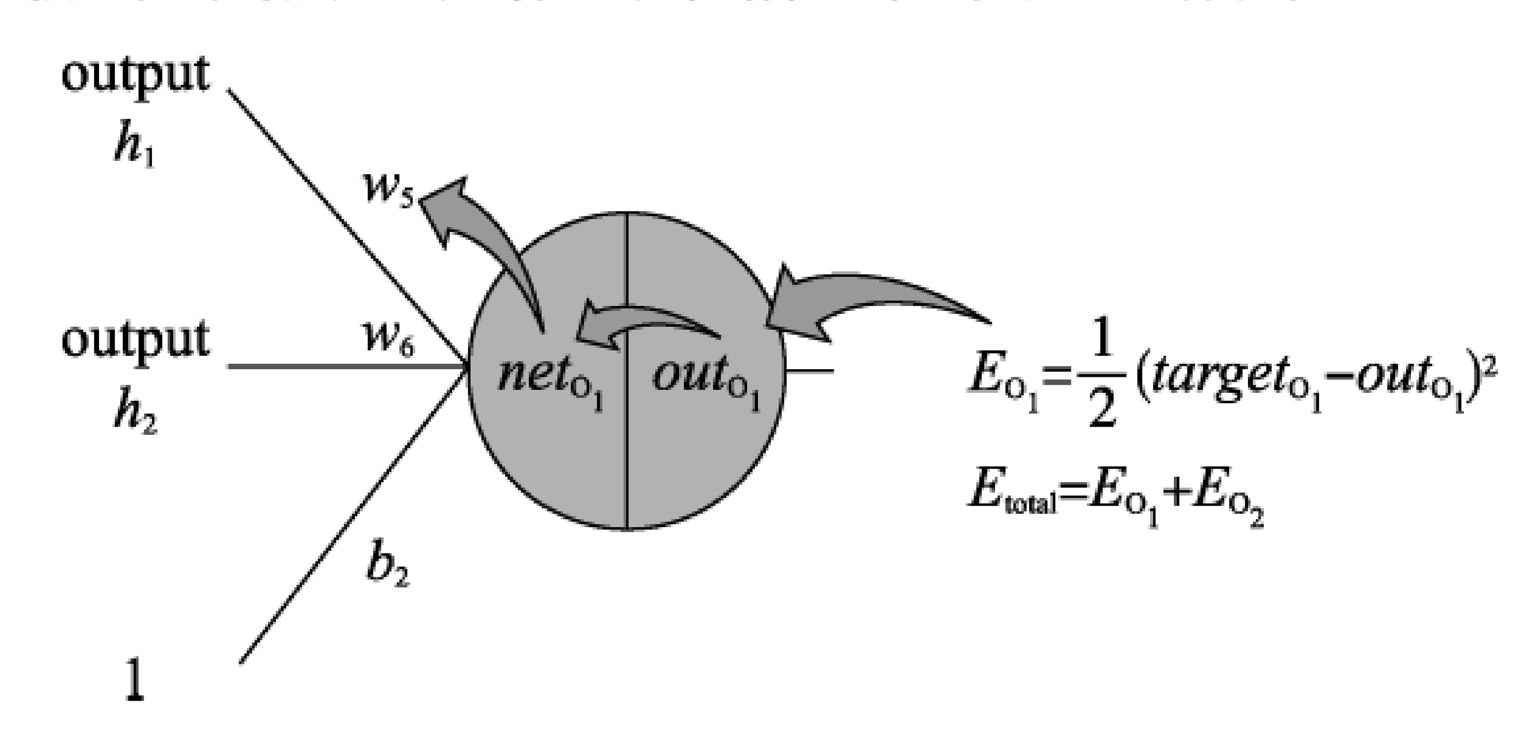


图 7-4　E_{total} 对 w_5 求偏导数

我们需要找出这个方程中的每一部分。首先，求总误差相对于输出的变化。

$$E_{\text{total}} = \frac{1}{2}(target_{o_1} - out_{o_1})^2 + \frac{1}{2}(target_{o_2} - out_{o_2})^2$$

$$\frac{\partial E_{\text{total}}}{\partial out_{o_1}} = 2 \times \frac{1}{2}(target_{o_1} - out_{o_1})^{2-1} \times (-1) + 0$$

$$\frac{\partial E_{\text{total}}}{\partial out_{o_1}} = -(target_{o_1} - out_{o_1}) = -(0.01 - 0.75136507) = 0.74136507$$

当对 out_{o_1} 取总误差的偏导数时，$\frac{1}{2}(target_{o_2} - out_{o_2})^2$ 变为 0，因为 out_{o_1} 不会影响它，这意味着我们正在取一个常数为 0 的导数。

接下来，求 o_1 的输出相对于其总净投入量的变化。

逻辑函数的偏导数是输出乘以（1 减去输出），得

$$out_{o_1} = \frac{1}{1 + e^{-net_{o_1}}}$$

$$\frac{\partial out_{o_1}}{\partial net_{o_1}} = out_{o_1}(1 - out_{o_1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

最后，求 o_1 的总净输入相对于 w_5 的变化。

$$net_{o_1} = w_5 \times out_{h_1} + w_6 \times out_{h_2} + b_2 \times 1$$

$$\frac{\partial net_{o_1}}{\partial w_5} = 1 \times out_{h_1} \times w_5^{(1-1)} + 0 + 0 = out_{h_1} = 0.593269992$$

把它们放在一起，得

$$\frac{\partial E_{\text{total}}}{\partial w_5} = \frac{\partial E_{\text{total}}}{\partial out_{o_1}} \times \frac{\partial out_{o_1}}{\partial net_{o_1}} \times \frac{\partial net_{o_1}}{\partial w_5}$$

$$\frac{\partial E_{\text{total}}}{\partial w_5} = 0.74136507 \times 0.186815602 \times 0.593269992 = 0.082167041$$

为了减少误差，我们从当前权重中减去这个值（可选地乘以某个学习率 eta，我们将其设置为 0.5），得

$$w_5^+ = w_5 - \eta \times \frac{\partial E_{\text{total}}}{\partial w_5} = 0.4 - 0.5 \times 0.082167041 = 0.35891648$$

重复这个过程可以获得新的权重 w_6、w_7 和 w_8，得

$$\omega_6^+ = 0.408666186$$

$$\omega_7^+ = 0.511301270$$

$$\omega_8^+ = 0.561370121$$

在将新权重引入隐藏层神经元后，才执行神经网络中的实际更新（即当继续使用下面的反向传播算法时，我们是使用原始权重，而不是使用更新后的权重）。

接下来，我们将通过计算 w_1、w_2、w_3 和 w_4 的新值来继续向后传递。

从大的方面来说，以下是我们需要弄清楚的。

$$\frac{\partial E_{\text{total}}}{\partial w_1}=\frac{\partial E_{\text{total}}}{\partial out_{h_1}}\times\frac{\partial out_{h_1}}{\partial net_{h_1}}\times\frac{\partial net_{h_1}}{\partial w_1}$$

E_{total} 对 w_1 求偏导数如图 7-5 所示。

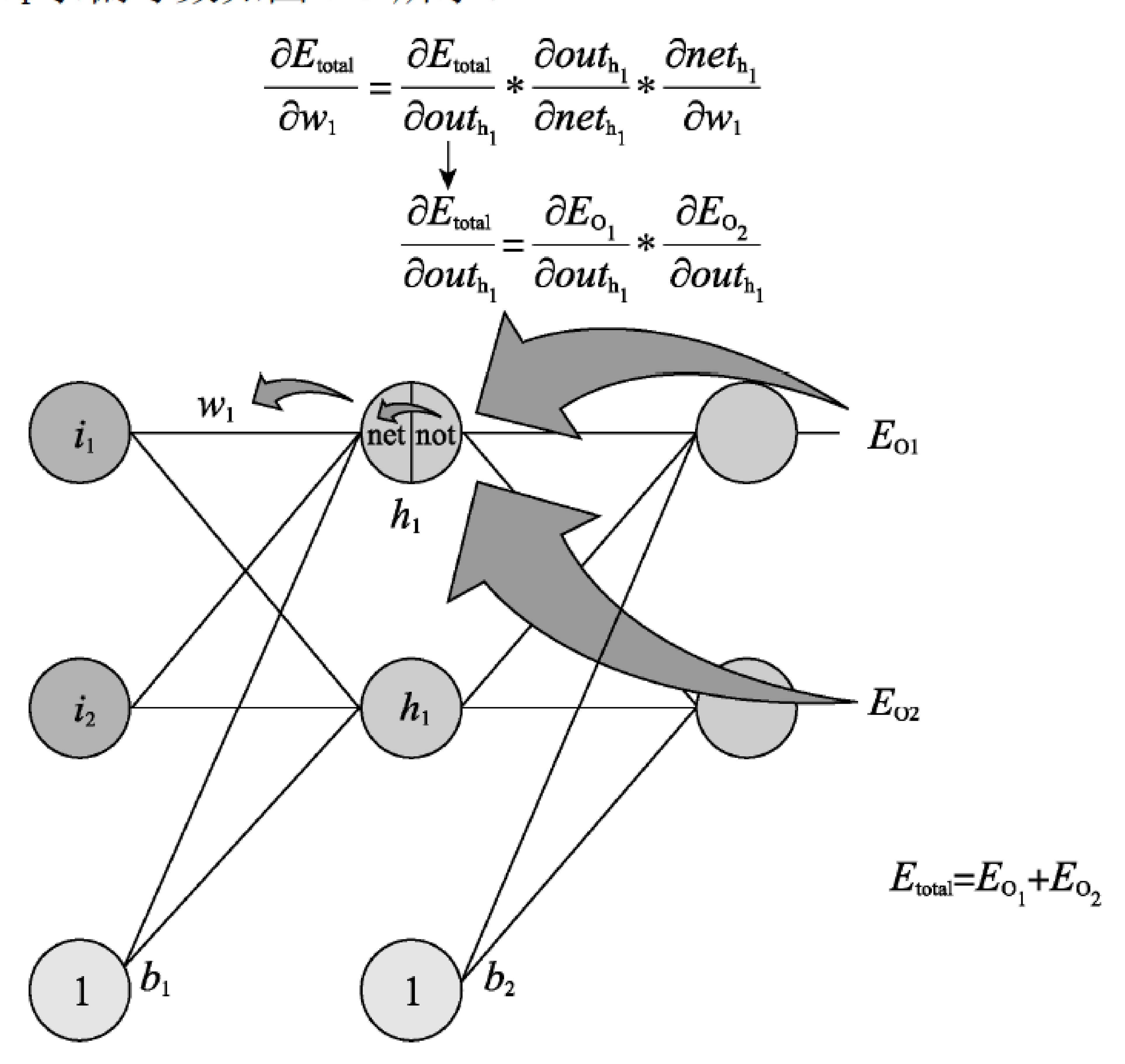


图 7-5　E_{total} 对 w_1 求偏导数

我们将使用与处理输出层类似的过程，但略有不同，以说明每个隐藏层神经元的输出对多个输出神经元的输出贡献（并因此产生误差）。我们知道 out_{h_1} 同时影响 out_{o_1} 和 out_{o_2}，因此，$\frac{\partial E_{\text{total}}}{\partial out_{h_1}}$ 需要考虑它对两个输出神经元的影响，得

$$\frac{\partial E_{\text{total}}}{\partial out_{h_1}}=\frac{\partial E_{o_1}}{\partial out_{h_1}}+\frac{\partial E_{o_2}}{\partial out_{h_1}}$$

从 $\frac{\partial E_{o_1}}{\partial out_{h_1}}$ 开始：

$$\frac{\partial E_{o_1}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial net_{o_1}} \times \frac{\partial net_{o_1}}{\partial out_{h_1}}$$

使用之前计算的值来计算 $\frac{\partial E_{o_1}}{\partial net_{o_1}}$：

$$\frac{\partial E_{o_1}}{\partial net_{o_1}} = \frac{\partial E_{o_1}}{\partial out_{o_1}} \times \frac{\partial out_{o_1}}{\partial net_{o_1}} = 0.74136507 \times 0.186815602 = 0.138498562$$

且 $\frac{\partial net_{o_1}}{\partial out_{h_1}}$=$w_5$，推导过程如下：

$$net_{o_1} = w_5 \times out_{h_1} + w_6 \times out_{h_2} + b_2 \times 1$$

$$\frac{\partial net_{o_1}}{\partial out_{h_1}} = w_5 = 0.40$$

代入这些值，得

$$\frac{\partial E_{o_1}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial net_{o_1}} \times \frac{\partial net_{o_1}}{\partial out_{h_1}} = 0.138498562 \times 0.40 = 0.055399425$$

对于 $\frac{\partial E_{o_2}}{\partial out_{h_1}}$，按照相同的过程，得

$$\frac{\partial E_{o_2}}{\partial out_{h_1}} = -0.019049119$$

因此，$\frac{\partial E_{total}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial out_{h_1}} + \frac{\partial E_{o_2}}{\partial out_{h_1}} = 0.055399425 + (-0.019049119) = 0.036350306$

现在有 $\frac{\partial E_{total}}{\partial out_{h_1}}$，我们需要计算出 $\frac{\partial out_{h_1}}{\partial net_{h_1}}$，然后为每个权重计算出 $\frac{\partial net_{h_1}}{\partial w}$，得

$$out_{h_1} = \frac{1}{1 + e^{-net_{h_1}}}$$

$$\frac{\partial out_{h_1}}{\partial net_{h_1}} = out_{h_1}(1 - out_{h_1}) = 0.59326999 \times (1 - 0.59326999) = 0.241300709$$

计算总的网络输入到 h_1 对于 w_1 的偏导数，得

$$net_{h_1} = w_1 \times i_1 + w_3 \times i_2 + b_1 \times 1$$

$$\frac{\partial net_{h_1}}{\partial w_1} = i_1 = 0.05$$

把这些值放在一起，得

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} \times \frac{\partial out_{h_1}}{\partial net_{h_1}} \times \frac{\partial net_{h_1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 \times 0.241300709 \times 0.05 = 0.000438568$$

更新 w_1：

$$w_1^+ = w_1 - \eta \times \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 \times 0.000438568 = 0.149780716$$

对 w_2、w_3 和 w_4 重复此操作，得

$$\omega_2^+ = 0.19956143$$

$$\omega_3^+ = 0.24975114$$

$$\omega_4^+ = 0.29950229$$

最后，我们已经更新了所有的权重。当最初输入 0.05 和 0.1 时，网络的误差为 0.298371109。在第一轮反向传播后，总误差降至 0.291027924。下降的幅度可能看起来并不多，但是在重复此过程 10 000 次后，误差直线下降到 0.0000351085。此时，再输入 0.05 和 0.1，两个输出神经元输出 0.015912196（和 0.01 的目标值相比较）和 0.984065734（和 0.99 的目标值相比较）。

7.2　卷积神经网络

随着深度神经网络技术的成熟和发展，识别图像往往采用层数很深的神经网络。

输入层和隐藏层之间是通过权值连接起来的，如果把输入层和隐藏层的神经元全部连接起来，那么权值数量有点太多了。例如，对于一幅 1000×1000 大小的图像，输

入层的神经元个数就是像素点之和，个数为 1000×1000 个。再假设和这个输入层连接的隐藏层的神经元个数为 1000 000 个，那么权值数量就是 1000×1000×1000 000。

对于给定的输入图片，用一个卷积核处理这张图片，也就是说一个卷积核处理整张图，所以权重是一样的，这称为权值共享。卷积层提取出特征，再进行组合，形成更抽象的特征，最后形成对图片对象的描述特征。

卷积神经网络（Convolutional Neural Network，CNN）具有独特的结构，旨在模仿真实动物大脑的运转方式，而不是让每层中的每个神经元连接到下一层中的所有神经元（多层感知器），神经元以三维结构排列，以便考虑不同神经元之间的空间关系。

卷积神经网络一般采用卷积层与采样层交替设置，即一个卷积层后接一个采样层，采样层后接一个卷积层。Java 实现代码如下：

```
DataSet dataSet = new DataSet("dataSet/data.ds", 0.3);
System.out.println(dataSet.getTrainSize());      //训练样本大小
//定义网络结构
Cnn cnn = new CnnBuilder(50)                      //取样的批大小
        .setInputLayer(new Size(28, 28))          //输入层
        .addConvolutionalLayer(6, new Size(5, 5)) //卷积层
        .addSimpleLayer(new Size(2, 2))           //采样层
        .addConvolutionalLayer(12, new Size(5, 5))
        .addSimpleLayer(new Size(2, 2))
        .setOutputLayer(10)        //输出层,识别 10 个数字,所以是 10 个神经元
        .build();
long now = System.currentTimeMillis();
cnn.train(dataSet, 3);             //迭代 3 次,适当增加迭代次数可以提高精度
System.out.println("cost:" + (System.currentTimeMillis() - now));
                                   //花费时间
cnn.saveModel("demo.model");       //保存模型文件
```

训练需要较长的时间，在有的计算机上迭代 100 次需要 1 个多小时。

测试训练出的模型：

```
Cnn cnn = Cnn.readModel("demo.model"); //加载模型文件
final int[] testRight = { 0 };
```

```
    final int[] testCount = { 0 };
    DataSet dataSet = new DataSet("dataSet/data.ds", 0.3);

    dataSet.testRecordForEach(record -> {
       if (cnn.test(record)) {
          testRight[0]++;
       }
       testCount[0]++;
    });
    double testP = 1.0 * testRight[0] / testCount[0];
    logger.info("test precision " + testRight[0] + "/" + testCount[0] + "=" +
testP); //输出精度
```

实际使用卷积神经网络时，一个卷积层对应多个卷积核，每个卷积核计算出一个卷积特征图（Feature Map）。用一组卷积核探测图像在同一层次不同基上的描述。例如，一个卷积核探测水平边界特征，另一个卷积核探测垂直边界特征。十字架是图像中这两个卷积核都活跃的区域。

使用一个图像测试不同的卷积核。为了使用 PIL 模块读取图像文件，需要安装 Pillow 模块，实现代码如下：

```
    >pip3 install Pillow
```

测试二维卷积，实现代码如下：

```
    from PIL import Image
    import numpy as np
    from scipy import signal as sg
    def np from img(fname):
       return np.asarray(Image.open(fname), dtype=np.float32)
    def save as img(ar, fname):
       Image.fromarray(ar.round().astype(np.uint8)).save(fname)
    def norm(ar):
       return 255.*np.absolute(ar)/np.max(ar)
    img = np from img('img/portal.png')
    save as img(norm(sg.convolve(img, [[1.],
                                      [-1.]])),
              'img/portal-h.png')
    save as img(norm(sg.convolve(img, [[1., -1.]])),
              'img/portal-v.png')
```

对于印刷体，可以用感知直线的卷积核参数。展平卷积特征图，如果有 10 个大小为 5×5 的特征图，那么将得到一个具有 250 个值的图层，然后与 MLP 没有任何区别，将所有这些人工神经元通过权重连接到所有在下一层中的人工神经元。

如何创建卷积层中使用的过滤器？可以使用反向传播算法训练它，就像训练 MLP 一样。在训练期间，可以根据网络参数优化某些损失函数。这样做，事实证明，边缘或弯曲特征会导致比随机特征更低的误差。

计算特征图的大小。使用步幅为 1、大小为 3×3 的卷积核。

计算特征图输出尺寸的公式为

$$\frac{N-K}{S}+1$$

在前面的例子中，$N=7$，$K=3$，$S=1$，根据公式计算出输出尺寸为 5。

SciPy（https://www.scipy.org）是一款免费的开源 Python 库，用于科学计算和技术计算。使用 scipy.signal. convolve 方法可以计算一维卷积。例如：

```
import numpy as np
import scipy.signal
x=np.array([1, 0, 2, 3, 0, 1, 1])     #输入
h=np.array([2,1,3])                   #卷积核
scipy.signal.convolve(x,h)
```

输出结果：

```
array([ 2,  1,  7,  8,  9, 11,  3,  4,  3])
```

要手动计算一维卷积，可以在输入上滑动内核，求逐元素乘法并对它们求和。操作前，先把卷积核中的数组反转过来，然后对应元素相乘。

使用 scipy.signal.convolve2d 方法可以计算二维卷积。例如：

```
import scipy.signal
image = [[1, 2, 3, 4, 5, 6, 7],
        [8, 9, 10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19, 20, 21],
        [22, 23, 24, 25, 26, 27, 28],
        [29, 30, 31, 32, 33, 34, 35],
        [36, 37, 38, 39, 40, 41, 42],
```

```
        [43, 44, 45, 46, 47, 48, 49]]
filter kernel = [[-1, 1, -1],
                [-2, 3, 1],
                [2, -6, 0]]
res = scipy.signal.convolve2d(image, filter kernel,
                          mode='same', boundary='fill', fillvalue=0)
print(res)
```

输出结果：

```
[[  -2   -8   -7   -6   -5   -4   28]
 [   3   -7  -10  -13  -16  -19   14]
 [ -18  -28  -31  -34  -37  -40    0]
 [ -39  -49  -52  -55  -58  -61  -14]
 [ -60  -70  -73  -76  -79  -82  -28]
 [ -81  -91  -94  -97 -100 -103  -42]
 [-101  -61  -63  -65  -67  -69  -57]]
```

为了手工计算，可以先反转卷积核 filter_kernel 为：

```
[[0, -6, 2],
 [1, 3, -2],
 [-1, 1, -1]]
```

然后对应元素相乘，例如左上角第一个结果的计算方法为

$$3\times1+(-2)\times2+1\times8+(-1)\times9=-2$$

个体感觉神经元的感受野是感觉空间（如身体表面或视野）的特定区域，感受野中的刺激将改变该神经元的发射。该区域可以是耳蜗中的纤毛或皮肤、视网膜、舌头或动物身体的其他部分。

使用 tf.contrib.receptive_field 可以轻松计算卷积神经网络的感受野参数，还可以了解输出特征所依赖的输入图像区域的大小。更好的是，使用库计算的参数，可以轻松找到用于计算每个卷积网络特征的精确图像区域。要调用的主要函数是 compute_receptive_field_from_graph_def()，它将返回感受野、水平和垂直方向的有效步幅及有效填充。使用函数 my_model_construction()构造模型，则可以按如下方式使用库。

```
import tensorflow as tf
```

```
    #构造图
    g = tf.Graph()
    with g.as_default():
      images = tf.placeholder(tf.float32, shape=(1, None, None, 3), name='input
image')
      my_model_construction(images)

    #计算感受野参数
    rf_x, rf_y, eff_stride_x, eff_stride_y, eff_pad_x, eff_pad_y = \
      tf.contrib.receptive_field.compute_receptive_field_from_graph_def( \
        g.as_graph_def(), 'input_image', 'my_output_endpoint')
```

rf_x = rf_y = 3039，eff_stride_x = eff_stride_y = 32，eff_pad_x = eff_pad_y = 1482，这意味着在节点 InceptionResnetV2/Conv2d_7b_1x1/Relu 输出的每个特征都是从一个 3039×3039 大小的区域计算的。此外，通过使用以下表达式：

```
    center_x = -eff_pad_x + feature_x*eff_stride_x + (rf_x - 1)/2
    center_y = -eff_pad_y + feature_y*eff_stride_y + (rf_y - 1)/2
```

可以计算输入图像中用于计算位于[feature_x,feature_y]处的输出特征区域的中心。例如，位于层 InceptionResnetV2/Conv2d_7b_1x1/Relu 的输出在[0,2]处的特征在原始图像中居中的位置为[37,101]。

可以直接从图形.pbtxt（protobuf）文件计算感受野参数。假设有 graph.pbtxt 文件并想要计算其感知字段参数，唯一的先决条件是安装 google/protobuf。如果使用 TensorFlow，则可能已安装 google/protobuf。

运行如下命令：

```
    cd python/util/examples
    python compute_rf.py \
      --graph_path /path/to/graph.pbtxt \
      --output_path /path/to/output/rf_info.txt \
      --input_node my_input_node \
      --output_node my_output_node
```

如果不知道如何生成图形 protobuf 文件，可以查看 write_inception_resnet_v2_graph.py 脚本，该脚本显示如何为 inception-resnet-v2 模型保存它。

```
cd python/util/examples
python write inception resnet v2 graph.py --graph dir /tmp --graph filename
graph.pbtxt
```

上述命令会将 inception-resnet-v2 图形 protobuf 文件写入/tmp/graph.pbtxt。

以下是使用此文件获取 inception-resnet-v2 模型感受野参数的命令。

```
cd python/util/examples
python compute rf.py \
  --graph path /tmp/graph.pbtxt \
  --output path /tmp/rf info.txt \
  --input node input image \
  --output_node InceptionResnetV2/Conv2d_7b_1x1/Relu
```

上述命令会将模型的感受野参数写入/tmp/rf_info.txt，如下所示。

```
Receptive field size (horizontal) = 3039
Receptive field size (vertical) = 3039
Effective stride (horizontal) = 32
Effective stride (vertical) = 32
Effective padding (horizontal) = 1482
Effective padding (vertical) = 1482
```

7.3 搭建深度学习开发环境

目前，很多深度学习框架底层采用 C++、C 或 Java 开发。本节先讲解使用集成开发环境 Eclipse-CDT 开发 C++或 C 应用。

7.3.1 使用 Cygwin 模拟环境

为了能够在 Windows 操作系统下使用 GUN 的 C++编译器，首先安装 Linux 模拟环境 Cygwin。可以从 Cygwin 的官方网站 http://www.cygwin.com/下载 Cygwin 的安装程序。

选择 Install from Internet，直接从 Internet 安装。使用网易镜像（http://mirrors.163.

com/cygwin/)。选择需要下载安装的组件包，如 gcc-core、gcc-g++、make、gdb 和 binutils。使用命令行安装相关组件：

```
    > Cygwin setup-x86 64.exe -q -P wget -P gcc-g++ -P make -P gcc-core -P gdb
-P binutils
```

在命令行输入如下命令，验证 C++编译器是否已经正确安装。

```
    > g++ -v
```

对于像深度学习框架 Darknet(https://github.com/pjreddie/darknet)这样包含 Makefile 的 C 语言源代码项目，可以直接导入 Eclipse-CDT。用 git 命令下载 Darknet：

```
    > git clone https://github.com/pjreddie/darknet
```

然后把 Darknet 源代码导入 Eclipse-CDT。

在 Eclipse-CDT 中可以直接调用 Darknet 训练好的模型。下载预先训练好的权重文件：

```
    > wget https://pjreddie.com/media/files/yolov3.weights
```

在 Eclipse-CDT 中运行检测器：

```
    >./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
```

这里，网络结构通过配置文件 cfg/yolov3.cfg 指定。

7.3.2 使用 CMake

编译工具 CMake（https://cmake.org/）使用 CMakeLists.txt 来生成 makefile 文件。CMake 通过在 CMakeLists.txt 文件中编写指令来控制项目，项目中的每个目录都应该有一个 CMakeLists.txt 文件。CMake 的好处在于，子目录中的 CMakeLists.txt 文件继承父目录中设置的属性，从而减少代码重复量。

在 Linux 操作系统下安装或升级 CMake，只需从 https://cmake.org/download/下载并安装 CMake 的较新版本。

```
    #cd /usr
    #sudo  wget  https://cmake.org/files/v3.8/cmake-3.8.2-Linux-x86 64.sh  -P
/usr/
```

```
#sudo chmod 755 /usr/cmake-3.8.2-Linux-x86 64.sh
#sudo ./cmake-3.8.2-Linux-x86_64.sh
```

在 Windows 操作系统下安装 CMake，首先下载 CMake。安装 CMake 后，把 cmake.exe 所在的路径增加到 PATH 环境变量。

在 Eclipse-CDT 中不能创建 CMake 项目，但可以导入 CMake 项目。可以这样做——假设名为“dlib”的 CMake 项目源代码位于 D:/javaworkspace/src/dlib 下，创建一个文件夹 D:/javaworkspace/build/dlib，切换到该文件夹下，并使用 Eclipse 生成器运行 CMake：

```
>cmake ../../src/dlib -G"Eclipse CDT4 - Unix Makefiles"
```

7.3.3 使用 Keras

在安装 Keras 之前，需要先安装后端引擎——可以安装 TensorFlow 或 CNTK 作为后端引擎。下面讲解 TensorFlow 后端引擎的安装方法。

在 Windows 操作系统中，TensorFlow 仅支持 64 位 Python 3.5.x 以上版本。当下载 Python 3.5.x 版本时，开发商为 Python 随附了 pip3 软件包管理器，以使用它来安装 TensorFlow。如果操作系统中没有安装低版本的 Python 2，则可以使用 pip 命令安装包。

通过 pip.ini 文件可以指定安装参数。例如：

```
[global]
index-url = http://mirrors.aliyun.com/pypi/simple
trusted-host = mirrors.aliyun.com
disable-pip-version-check = true
timeout = 120
[list]
format = columns
```

可以将 pip.ini 文件放置在%APPDATA%\pip\目录下。

使用 pip 安装 TensorFlow：

```
>pip install tensorflow==1.5 -i https://pypi.douban.com/simple/
```

验证安装是否成功：

```
import tensorflow;
print(tensorflow.__version__);
```

如果正确输出版本号 1.5.0，则说明安装成功。

然后安装 Keras：

```
>pip install keras
```

使用手写字符数据集测试。首先下载 mnist.npz 文件中的数据集。.npz 格式文件是一种压缩文件。其中包含了以变量命名的一些.npy 文件，这里为 x_train.npy、x_test.npy、y_train.npy、y_test.npy 这 4 个文件。x_train.npy 和 x_test.npy 中是神经网络的输入数据，y_train.npy、y_test.npy 中是神经网络的预期输出数据。

读取 y_test.npy 中的数据：

```
import numpy as np;
c = np.load( "d:/soft/mnist/y_test.npy" );
print(c);
```

输出结果：

```
[7 2 1 ... 4 5 6]
```

为了在 Java 中读/写.npy 和.npz 文件，可以使用 npy（https://github.com/JetBrains-Research/npy）包。

Keras 实现卷积神经网络分类的代码如下：

```
from __future__ import print_function
import keras
from keras.datasets import mnist
from keras.models import Sequential       #序列模型是一个线性的层次堆叠
from keras.layers import Dense, Dropout   #将要使用的两种类型的神经网络层
from keras.optimizers import RMSprop      #将要使用的优化器
batch_size = 128      #指定进行梯度下降时每个批次包含的样本数
num_classes = 10      #标签为 0~9 共 10 个类别
epochs = 20           #时期
import numpy as np
path = 'd:/soft/mnist.npz'
f = np.load(path)
x_train, y_train = f['x_train'], f['y_train']
```

```
x_test, y_test = f['x_test'], f['y_test']
f.close()

#输入图像大小为 28×28,所以是 784 个特征
x train = x train.reshape(60000, 784).astype('float32')
x test = x test.reshape(10000, 784).astype('float32')
x train /= 255
x test /= 255
print(x train.shape[0], 'train samples')
print(x test.shape[0], 'test samples')

#keras 要求标签格式为 One-hot 编码
y_train = keras.utils.to_categorical(y_train, num_classes)#对标签进行 One-hot 编码
y_test = keras.utils.to_categorical(y_test, num_classes) #对标签进行 One-hot 编码
model = Sequential()
#输入是长度为 784 的一维向量
model.add(Dense(512, activation='relu', input shape=(784,)))
#Dense 就是全连接层
#添加 Dropout 层
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))
model.summary()
model.compile(loss='categorical_crossentropy',     #使用交叉熵损失函数
            optimizer=RMSprop(),                   #使用 RMSprop 优化器
            metrics=['accuracy'])                  #报告准确性

history = model.fit(x_train, y_train,              #使用训练集训练模型
                 batch size=batch size,
                 epochs=epochs,                    #训练的轮数
                 verbose=1,
                 validation data=(x test, y test))
#在测试集上评估训练好的模型
score = model.evaluate(x test, y test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

7.3.4　安装 TensorFlow

在 Windows 操作系统下安装 TensorFlow。如果需要，则升级 pip：

```
>python -m pip install --upgrade pip
```

安装 TensorFlow 的指定版本 1.8.0：

```
>pip install tensorflow==1.8.0
```

如果下载软件包超时，则可以指定超时时限：

```
>pip install tensorflow==1.8.0 --default-timeout=1000
```

测试版本：

```
import tensorflow as tf
print(tf.__version__)
```

在 Ubuntu 操作系统下通过编译源代码的方法安装 TensorFlow。

Bazel 是编译 TensorFlow 的工具软件。Bazel 将整个构建分解为独立的步骤，称为动作。每个动作都有输入/输出名称、命令行和环境变量。每个动作明确声明所需输入和预期输出。

```
#wget https://github.com/bazelbuild/bazel/releases/download/0.13.0/bazel-0.13.0-installer-linux-x86_64.sh
#chmod +x bazel-0.13.0-installer-linux-x86_64.sh
#./bazel-0.13.0-installer-linux-x86_64.sh -user
```

安装 JDK 8，执行如下命令：

```
#sudo apt-get install openjdk-8-jdk
```

安装 Python 2.7，执行如下命令：

```
#sudo apt-get install python-pip python-numpy swig python-dev
#sudo pip install wheel
```

安装 Python 3，执行如下命令：

```
#sudo apt-get install python3-pip python3-numpy swig python3-dev
#sudo pip3 install wheel
```

取得源代码：

```
#git clone https://github.com/tensorflow/tensorflow.git
```

在本地源代码库的根目录下更新源代码：

```
#git fetch origin
```

通过 bazel 配置：

```
#./configure
```

构建 TensorFlow 包：

```
#bazel build --jobs 1 --config=monolithic tensorflow/tools/pip_package:
build_pip_package
```

7.3.5 安装 TensorFlow 的 Docker 容器

安装 Docker：

```
$sudo apt install docker.io
```

启动 Docker 服务：

```
$sudo systemctl start docker
```

列出镜像：

```
$sudo docker images
```

CentOS 下安装镜像：

```
#yum -y install docker-io
```

启动服务：

```
#service docker start
```

查看服务状态：

```
#service docker status
```

查看镜像：

```
https://hub.docker.com/r/tensorflow/tensorflow/tags/
```

得到镜像：

```
#docker pull tensorflow/tensorflow:1.8.0-py3
```

其中 1.8.0-py3 为标签名。

列出镜像：

```
#docker image ls
```

运行镜像：

```
#docker run -it docker.io/tensorflow/tensorflow:1.8.0-py3  /bin/bash
```

查看正在运行的容器：

```
#docker ps
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES
0c8e82bc5c22  docker.io/tensorflow/tensorflow:1.8.0-py3
   "/bin/bash"  2 hours ago  Up 2 hours  6006/tcp, 8888/tcp  priceless_hawking
```

查看指定容器的信息：

```
#docker inspect 0c8
```

停止指定容器：

```
#docker stop 0c8
```

停止 Docker 服务：

```
$sudo systemctl disable docker.service
```

停止所有的 Docker 容器：

```
$sudo docker ps -a -q | xargs -n 1 -P 8 -I {} docker stop {}
```

为了完全卸载 Docker，需要确定已经安装的包：

```
$dpkg -l | grep -i docker
```

卸载已经安装的包：

```
$sudo apt-get purge -y docker-engine docker docker.io docker-ce
$sudo apt-get autoremove -y --purge docker-engine docker docker.io docker-ce
```

上述命令不会删除主机上的镜像、容器、卷或用户创建的配置文件。如果要删除所有镜像、容器和卷，运行以下命令：

```
$sudo rm -rf /var/lib/docker
$sudo rm /etc/apparmor.d/docker
```

```
$sudo groupdel docker
$sudo rm -rf /var/run/docker.sock
```

7.3.6 使用 TensorFlow

基本上，所有 TensorFlow 代码都包含两个重要部分。

第 1 部分：构建计算图来表示计算的数据流。

第 2 部分：运行会话来执行计算图中的操作。

首先创建计算图，即想要对数据执行的操作，然后使用会话单独运行它。

TensorFlow 程序使用称为张量（Tensor）的数据结构来表示所有数据。计划用于模型的任何类型数据都可以存储在 Tensor 中。简而言之，张量是一个多维数组（零维张量：标量；一维张量：向量；二维张量：矩阵等）。 因此，TensorFlow 只是指计算图中张量的流动。

计算图是一系列排列成节点图的 TensorFlow 操作。基本上，这意味着图形只是表示模型中操作的节点布局。例如，函数 $f(x,y)=x^2y+y+2$ 在 TensorFlow 中的计算图如图 7-6 所示。

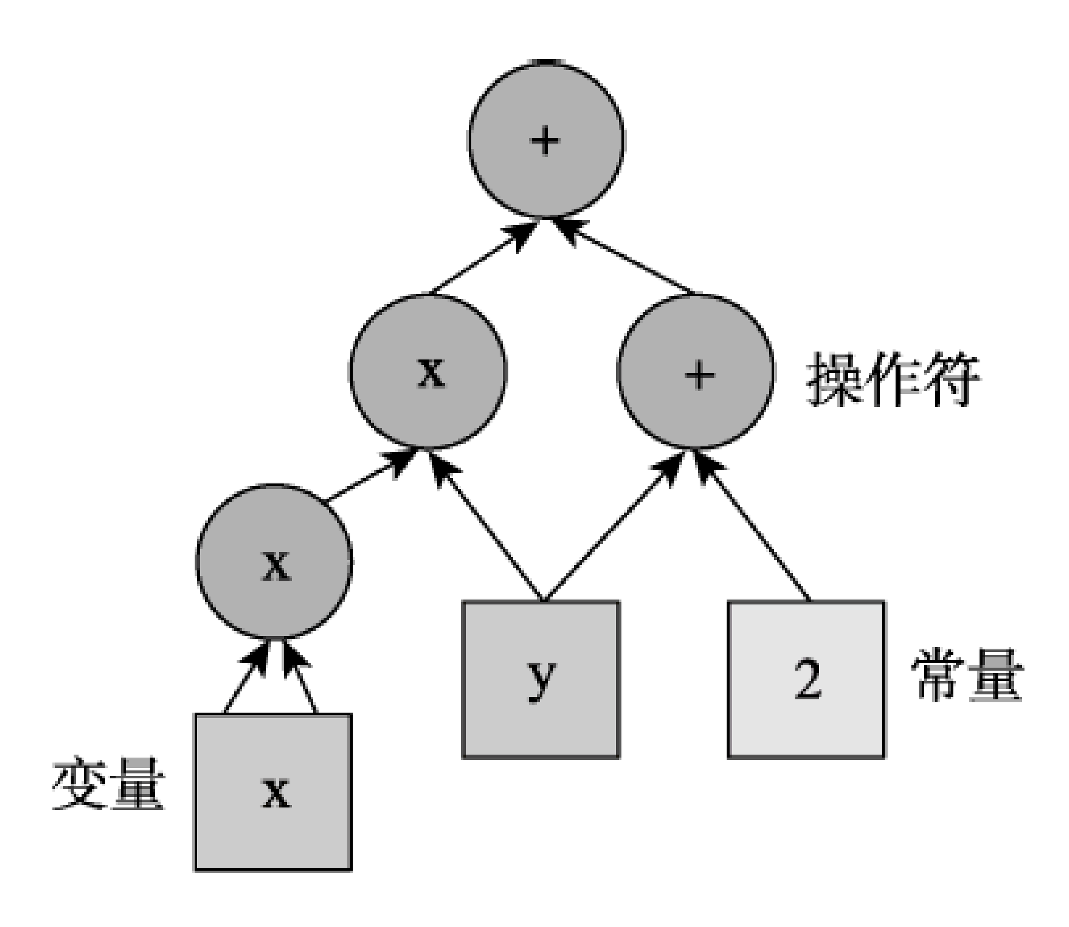


图 7-6　函数 $f(x,y)$ 的计算图

下面让我们从一个基本的算术操作开始，演示一个计算图。该代码使用 TensorFlow 添加两个值，a=2 和 b=3。为此，我们需要调用 tf.add()。tf.add()有 3 个参数 a、b 和 name，其中 a 和 b 为要加在一起的值，name 为操作名称，即与计算图上的加法节点相关联的

名称。

```
import tensorflow as tf
a = 2
b = 3
c = tf.add(a, b, name='Add')
print(c)
```

输出：

```
Tensor("Add:0", shape=(), dtype=int32)
```

在上述这段代码中，我们用“Python 名称”生成了 3 个变量——*a*、*b* 和 *c*。这里，*a* 和 *b* 是 Python 变量，因此没有“TensorFlow 名称”，而 *c* 是一个带有“TensorFlow 名称”的张量。

要计算任何内容，必须在会话中启动图形。从技术上讲，会话将图形操作放置在诸如 CPU 或 GPU 等的硬件上，并提供执行它们的方法。在本示例中，需要运行该图并获取张量 *c* 的值。以下代码将创建一个会话并通过运行张量 *c* 来执行该图。

```
sess = tf.Session()
print(sess.run(c))
sess.close()
```

上述代码创建一个 Session 对象（分配给变量 sess），然后调用 sess 的 run 方法以运行足够的计算图来评估 *c*，这意味着，它只运行图的这个部分来获得 *c* 的值。在这个简单的例子中，它运行整个图。在会话结束时关闭会话，这是使用上述代码中的最后一行完成的。

以下代码执行相同的操作且更常用。唯一的区别是，不需要在结束时自动关闭会话。

```
with tf.Session() as sess:
    print(sess.run(c))
```

通过张量的名称来运行计算图，代码如下：

```
import tensorflow as tf
a = 2
b = 3
c = tf.add(a, b, name='Add')
```

```
with tf.Session() as sess:
    print(sess.run('Add:0'))
```

为了方便调试，可以调用打印操作输出张量的值，实现代码如下：

```
import tensorflow as tf
#注册默认的会话
sess = tf.InteractiveSession()
#想要打印出值的张量
a = tf.constant([1.0, 3.0])
#添加打印操作
a = tf.Print(a, [a], message="This is a: ")
#使用张量 a 往计算图添加更多的元素
b = tf.add(a, a)
#执行计算图
b.eval()
```

输出结果：

```
This is a: [1 3]
```

使用 tf.constant 可以简单地创建一个常数张量，它可接受 5 个参数。例如：

```
tf.constant(value, dtype=None, shape=None, name='Const', verify_shape=False)
```

以下是一个非常简单的例子——创建 *a*、*b* 两个常量并将它们加在一起。

```
#常量张量可以简单地用一个值来定义
a = tf.constant(2)
b = tf.constant(3)

#运行默认的图
with tf.Session() as sess:
    print("a=2, b=3")
    print("Addition with constants: %i" % sess.run(a+b))
```

常量也可以用不同类型（整型、浮点型等）和形状（向量、矩阵等）来定义。假设有一个 32 位浮点类型的常量和另一个形状为 2×2 的常量，实现代码如下：

```
s = tf.constant(2.3, name='scalar', dtype=tf.float32)
m = tf.constant([[1, 2], [3, 4]], name='matrix')
#在会话中运行计算图
with tf.Session() as sess:
```

```
    print(sess.run(s))
    print(sess.run(m))
```

变量是输出其当前值的有状态节点。这意味着变量可以在一个计算图的多次执行中保留其值。变量包括以下多个有用的功能。

- 在训练期间和之后，可以把变量保存到磁盘。这样可以让不同公司和团体的人员进行协作，因为他们可以保存、恢复并将模型参数发送给其他人。
- 默认情况下，梯度更新（用于所有神经网络）将应用于图形中的所有变量。事实上，变量是想要调整的事物，以便将损失降至最低。

以上这两个特征使变量适合用作网络参数（即权重和偏差）。

变量和常量之间有以下两个主要区别。

- 常量的值不会改变。但我们通常需要更新网络参数，这就是变量发挥作用的地方。
- 常量存储在图形定义中，这使得内存非常紧张。换句话说，具有数百万条目的常量会使图形显示速度变得更慢且资源密集。

创建变量是一种操作，可以在会话中执行这些操作并获取操作的输出值。要创建一个变量，应该使用 tf.Variable()。例如：

```
#创建一个变量
w = tf.Variable(<initial-value>, name=<optional-name>)
```

就如同在大多数编程语言中一样，变量在使用之前需要初始化。TensorFlow 虽然不是一种语言，但也不例外。要初始化变量，我们必须调用一个变量初始值设定项操作并在会话中运行该操作。这是一次性初始化所有变量的最简单方法。

创建 a、b 两个变量并将它们加在一起，实现代码如下：

```
#创建计算图
a = tf.get variable(name="A", initializer=tf.constant(2))
b = tf.get variable(name="B", initializer=tf.constant(3))
c = tf.add(a, b, name="Add")
#添加一个操作来初始化全局变量
init op = tf.global variables initializer()
#在会话中运行计算图
with tf.Session() as sess:
```

```
    #运行变量初始化器操作
    sess.run(init_op)
    #评估变量的值
    print(sess.run(a))
    print(sess.run(b))
    print(sess.run(c))
```

每次调用 tf.Variable()都会得到一个新的变量，不会检查名称冲突。但使用 tf.get_variable()时，则会检查名称冲突。使用 tf.Variable()的代码如下：

```
import tensorflow as tf
w_1 = tf.Variable(3,name="w_1")
w_2 = tf.Variable(1,name="w_1")
print(w_1.name)
print(w_2.name)
```

输出结果：

```
w_1:0
w_1_1:0
```

使用 tf.get_variable()的代码如下：

```
import tensorflow as tf
w_1 = tf.get_variable(name="w_1",initializer=1)
w_2 = tf.get_variable(name="w_1",initializer=2)
```

输出结果会报错：

```
ValueError: Variable w_1 already exists, disallowed. Did you mean to set
reuse=True or reuse=tf.AUTO_REUSE in VarScope?
```

变量通常用于神经网络中的权重和偏差。

- 权重通常使用 tf.truncated_normal_initializer()从正态分布初始化。
- 偏差通常使用 tf.zeros_initializer()从零初始化。

下面让我们看一个非常简单的例子——通过适当地初始化来创建权重和偏差变量。

为具有两个神经元的完全连接层创建权重和偏差矩阵，并将其与 3 个神经元的另一个图层一起创建。在这种情况下，权重和偏差变量的大小必须分别为 [2,3]和 3。

```
#创建计算图
```

```
    weights = tf.get variable(name="W", shape=[2,3], initializer=tf.truncated
normal initializer(stddev=0.01))
    biases = tf.get_variable(name="b", shape=[3], initializer=tf.zeros_initializer())
    #添加一个操作来初始化全局变量
    init op = tf.global variables initializer()
    #在会话中运行计算图
    with tf.Session() as sess:
        #运行变量初始化器操作
        sess.run(init op)
        #运行我们的操作
        W, b = sess.run([weights, biases])
        print('weights = {}'.format(W))
        print('biases = {}'.format(b))
```

输出结果：

```
    weights = [[-0.01064058 -0.0022427  -0.00125237]
     [ 0.00294374 -0.00230121 -0.01269217]]
    biases = [0. 0. 0.]
```

占位符比变量更基础，它只是在未来对数据进行分配的一个变量。占位符是其值在执行时被传入的节点。如果网络有依赖于某些外部数据的输入，并且不希望图形在开发时依赖于任何实际值，则占位符就是我们需要的数据类型。事实上，我们可以在没有任何数据的情况下构建图。因此，占位符不需要任何初始值。一个占位符只有一个数据类型（例如 float32）和一个张量形状，所以即使没有任何存储的值，图形仍然知道要计算什么。

创建占位符的示例如下。

```
    a = tf.placeholder(tf.float32, shape=[5])
    b = tf.placeholder(dtype=tf.float32, shape=None, name=None)
    X = tf.placeholder(tf.float32, shape=[None, 784], name='input')
    Y = tf.placeholder(tf.float32, shape=[None, 10], name='label')
```

使用占位符执行加法和乘法操作的示例如下。

```
    a = tf.placeholder(tf.int16)
    b = tf.placeholder(tf.int16)
    #定义一些操作
```

```
    add = tf.add(a, b)
    mul = tf.multiply(a, b)
    #启动默认的计算图
    with tf.Session() as sess:
        #用变量输入运行每个操作
        print("Addition with variables: %i" % sess.run(add, feed dict={a: 2, b: 3}))
        print("Multiplication with variables: %i" % sess.run(mul, feed dict={a:
2, b: 3}))
```

输出结果：

```
    Addition with variables: 5
    Multiplication with variables: 6
```

结合 numpy 使用的示例如下。

```
    import tensorflow as tf
    import numpy as np
    #创建一个 TensorFlow 常量
    const = tf.constant(2.0, name="const")
    #创建 TensorFlow 变量
    b = tf.placeholder(tf.float32, [None, 1], name='b')
    c = tf.Variable(1.0, name='c')
    #创建一些操作
    d = tf.add(b, c, name='d')
    e = tf.add(c, 2, name='e')
    a = tf.multiply(d, e, name='a')
    #设置变量初始化
    init op = tf.global variables initializer()
    #开始会话
    with tf.Session() as sess:
        #初始化变量
        sess.run(init op)
        #计算图的输出
        a out = sess.run(a, feed dict={b: np.arange(0, 10)[:, np.newaxis]})
        print("Variable a is {}".format(a_out))
```

输出结果：

```
    Variable a is [[ 3.]
     [ 6.]
```

```
 [ 9.]
 [12.]
 [15.]
 [18.]
 [21.]
 [24.]
 [27.]]
```

TensorFlow 解决 XOR 问题的神经网络，实现代码如下：

```
import tensorflow as tf
import numpy as np
tf.set_random_seed(777)      #为了可重现性
learning_rate = 0.1          #学习率
x data = [[0, 0],
        [0, 1],
        [1, 0],
        [1, 1]]
y data = [[0],
        [1],
        [1],
        [0]]
x data = np.array(x data, dtype=np.float32)
y data = np.array(y data, dtype=np.float32)

X = tf.placeholder(tf.float32, [None, 2])
Y = tf.placeholder(tf.float32, [None, 1])

W1 = tf.Variable(tf.random normal([2, 2]), name='weight1')
b1 = tf.Variable(tf.random normal([2]), name='bias1')
layer1 = tf.sigmoid(tf.matmul(X, W1) + b1) #一个隐藏层

W2 = tf.Variable(tf.random normal([2, 1]), name='weight2')
b2 = tf.Variable(tf.random normal([1]), name='bias2')
hypothesis = tf.sigmoid(tf.matmul(layer1, W2) + b2) #网络的输出值
#损失函数
cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1 - Y) *
                   tf.log(1 - hypothesis))
#设置优化器
```

```
train = tf.train.RMSPropOptimizer(learning rate=learning rate).minimize(cost)
#计算准确度
#如果 hypothesis>0.5,则为真;否则为假
predicted = tf.cast(hypothesis > 0.5, dtype=tf.float32)
#计算平均值
accuracy = tf.reduce_mean(tf.cast(tf.equal(predicted, Y), dtype=tf.float32))
#启动计算图
with tf.Session() as sess:
    #初始化 TensorFlow 变量
    sess.run(tf.global variables initializer())

    for step in range(10001):
        sess.run(train, feed_dict={X: x_data, Y: y_data})
        if step % 100 == 0:
            print(step, sess.run(cost, feed dict={
                 X: x data, Y: y data}), sess.run([W1, W2]))

    #准确性报告
    h, c, a = sess.run([hypothesis, predicted, accuracy],
                    feed_dict={X: x_data, Y: y_data})
    print("\nHypothesis: ", h, "\nCorrect: ", c, "\nAccuracy: ", a)

'''
Hypothesis:  [[ 0.01338218]
 [ 0.98166394]
 [ 0.98809403]
 [ 0.01135799]]
Correct:  [[ 0.]
 [ 1.]
 [ 1.]
 [ 0.]]
Accuracy:  1.0
```

只要元素的数量保持不变，就可以使用 tf.reshape()来改变 TensorFlow 张量的形状。这里将举 3 个例子来说明 tf.reshape()是如何工作的。

下面让我们从最初的 TensorFlow 常数张量形状 2×3×4 开始，数值范围为 1～24，所有数据类型都为 int32。

```
tf initial tensor constant = tf.constant(
[
    [
        [ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]
    ]
    ,
    [
        [13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24],
    ]
]
, dtype="int32"
)
```

在上述代码中，我们使用 tf.constant()创建了一个 2×3×4 的张量，数据类型为 int32，看到的数字是 1,2,3,4,…,24，将其分配给张量 tf_initial_tensor_constant。现在让我们打印出 tf_initial_tensor_constant 的 Python 变量来查看所拥有的内容。

```
print(tf_initial_tensor_constant)
```

输出结果：

```
Tensor("Const:0", shape=(2, 3, 4), dtype=int32)
```

可以看到，它是 TensorFlow 常量，形状为 2×3×4，数据类型为 int32。因为还没有在 TensorFlow 会话中运行它，所以即使将它定义为常量，似乎也没有值。这同样适用于我们即将创建的其他形状张量。

对于第一个例子，将形状为 2×3×4 的张量更改为形状为 2×12 的张量。

```
tf ex one reshaped tensor 2 by 12 = tf.reshape(tf initial tensor constant,
[2, 12])
```

这里使用函数 tf.reshape()，并传入 tf_initial_tensor_constant 和想要的新形状细节，然后将其分配给张量 tf_ex_one_reshaped_tensor_2_by_12。注意，元素的数量将保持不

变，因为 2×3×4 的取值是 24，2×12 的取值也是 24。

打印出张量 tf_ex_one_reshaped_tensor_2_by_12 来看看有什么变化。

```
print(tf_ex_one_reshaped_tensor_2_by_12)
```

输出结果：

```
Tensor("Reshape:0", shape=(2, 12), dtype=int32)
```

可以看到，它是 TensorFlow 张量，形状为 2×12，数据类型为 int32。输出还没有显示任何值，因为我们仍在构建 TensorFlow 图，还没有在 TensorFlow 会话中运行它。

对于第二个例子，将形状为 2×3×4 的张量更改为形状为 2×3×2×2 的张量。

```
tf ex two reshaped tensor 2 by 3 by 2 by 2 = tf.reshape(tf initial tensor constant, [2, 3, 2, 2])
```

这里使用函数 tf.reshape()，传入初始张量，并传入 2,3,2,2 指定形状，然后将它分配给张量 tf_ex_two_reshaped_tensor_2_by_3_ by_2_by_2。注意，元素的数量将保持不变，因为 2×3×4 的取值是 24，2×3×2×2 的取值也是 24。

打印出张量 tf_ex_two_reshaped_tensor_2_by_3_by_2_by_2 来看看什么变化。

```
print(tf_ex_two_reshaped_tensor_2_by_3_by_2_by_2)
```

输出结果：

```
Tensor("Reshape_1:0", shape=(2, 3, 2, 2), dtype=int32)
```

可以看到，它是 TensorFlow 张量，形状为 2×3×2×2（这是我们所期望的），数据类型为 int32。

对于第三个例子，将形状为 2×3×4 的 TensorFlow 张量更改为 24 个元素的向量。

```
tf ex tre reshaped tensor 1 by 24 = tf.reshape(tf initial tensor constant, [-1])
```

这里使用函数 tf.reshape()操作，传入初始张量，并传入“［-1］”。它的作用是将张量变平，所以得到的只是一个包含 24 个元素的列表。然后将它分配给张量 tf_ex_tre_reshaped_tensor_1_by_24。

打印出张量 tf_ex_tre_reshaped_tensor_1_by_24 来看看有什么变化。

```
print(tf_ex_tre_reshaped_tensor_1_by_24)
```

输出结果：

```
Tensor("Reshape_2:0", shape=(24,), dtype=int32)
```

可以看到，它是 TensorFlow 张量，形状为“(24,)”，这意味着它将是一个向量，数据类型为 int32。

创建 TensorFlow 张量后，下面开始运行计算图。

在会话中启动计算图：

```
sess = tf.Session()
```

初始化图中的所有全局变量：

```
sess.run(tf.global_variables_initializer())
```

接下来，打印出 4 个张量，以便了解 tf.reshape()的工作原理。

（1）打印出初始张量常数。

```
print(sess.run(tf_initial_tensor_constant))
```

输出结果：

```
[[[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]

 [[13 14 15 16]
  [17 18 19 20]
  [21 22 23 24]]]
```

可以看到，它是一个 $2\times3\times4$ 的张量，数字从 1 到 24，并且没有一个小数点，所以它们为 int32 类型数据。

（2）打印出第一个重塑张量。

```
print(sess.run(tf_ex_one_reshaped_tensor_2_by_12))
```

输出结果：

```
[[ 1  2  3  4  5  6  7  8  9 10 11 12]
 [13 14 15 16 17 18 19 20 21 22 23 24]]
```

可以看到，它是一个张量，里面有两个矩阵，第一个矩阵有 1 行 12 列，第二个矩阵也有 1 行 12 列，1～24 所有的元素都在其中。

（3）打印出第二个重塑张量。

```
print(sess.run(tf_ex_two_reshaped_tensor_2_by_3_by_2_by_2))
```

输出结果：

```
[[[[ 1  2]
   [ 3  4]]

  [[ 5  6]
   [ 7  8]]

  [[ 9 10]
   [11 12]]]


 [[[13 14]
   [15 16]]

  [[17 18]
   [19 20]]

  [[21 22]
   [23 24]]]]
```

可以看到，它是一个具有两个内部张量的张量，每个内部张量有 3 个 2×2 的矩阵。所以为两行两列；两行两列；两行两列，然后又是两行两列；两行两列；两行两列。

总的来说，我们可以看到形状为 2×3×2×2，所有的数字都在其中。

（4）打印出第三个重塑张量。

```
print(sess.run(tf_ex_tre_reshaped_tensor_1_by_24))
```

输出结果：

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

可以看到，它是一个有 24 个元素长的向量。只要元素的数量保持不变，就可以使用 tf.reshape()来改变 TensorFlow 张量的形状。

现在，我们拥有了所有必需的材料，可以开始构建带有一个隐藏层和 200 个隐藏

单元（神经元）的前馈神经网络。*h=ReLU*（*Wx*+*b*）TensorFlow 中的计算图如图 7-7 所示。

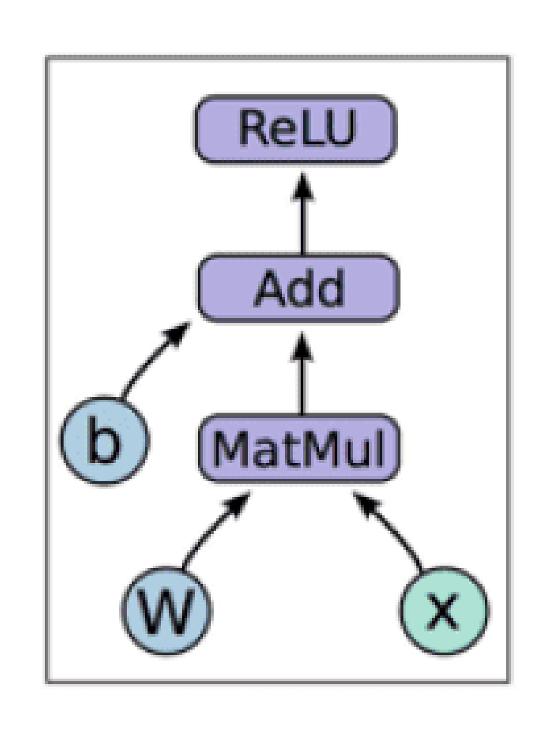


图 7-7　*h=ReLU*（*Wx*+*b*）的计算图

在现实世界的问题中，我们有成千上万的输入，这使得梯度下降的计算成本很高。这就是我们要将输入集分成几个尺寸为 B（称为小批量尺寸）的较短片段（称为小批量），并逐个输入它们的原因。我们把这种策略称为“随机梯度下降”（Stochastic Gradient Descent）。将大小为 B 的每个小批量馈送到网络，反向传播误差及更新参数（权重和偏差）的过程称为“迭代”。

我们通常使用占位符来输入，以便在上下文中没有任何实际值的情况下构建图。唯一的一点是，需要为输入选择适当的尺寸。在这里，我们有一个前馈神经网络，假设每个图像大小可为 784（类似于 MNIST 数据的 28×28 图像），输入占位符可以写为：

```
#创建输入占位符
X = tf.placeholder(tf.float32, shape=[None, 784], name="X")
```

你可能想知道为什么是 shape=[None, 784]?，这是因为如果我们需要在每次训练迭代中将 B 个大小为 784 的图像作为一个批次提供给网络，所以占位符为 shape = [B,784]。而将占位符的形状定义为 [None,784]，意味着我们可以提供任何数量的大小为 784 的图像（不一定是 B 个图像）。这在评估时特别有用，我们需要将所有验证或测试图像提供给网络，并计算出所有验证或测试图像的性能指标。

接下来，我们来看看网络参数 *W* 和 *b*。正如上面的变量部分所解释的那样，它们必须被定义为变量。由于在 TensorFlow 中默认情况下，渐变更新将应用于图形变量，

因此变量需要初始化。

一般来说，权重（W）是随机初始化的，它是正态分布中最简单的形式，例如零均值和标准差为 0.01 的正态分布。偏差（b）可以初始化为小的常数值，例如 0。

由于输入维数为 784，并且有 200 个隐藏单元，因此权重矩阵的大小为[784,200]。我们还需要 200 个偏差，每个隐藏单元一个。实现代码如下：

```
    #创建从 N(0, 0.01)随机初始化的权重矩阵
    weight_initer = tf.truncated_normal_initializer(mean=0.0, stddev=0.01)
    W = tf.get_variable(name="Weight", dtype=tf.float32, shape=[784,200],
initializer=weight_initer)

    #创建大小为 200 的偏差向量,全部初始化为 0
    bias_initer =tf.constant(0., shape=[200], dtype=tf.float32)
    b = tf.get_variable(name="Bias", dtype=tf.float32, initializer=bias_initer)
```

我们必须将输入 $X_{[None,784]}$和权重矩阵 $W_{[784,200]}$ 相乘，这个操作给出大小为[None,200]的张量，然后加上偏差向量 $b_{[200]}$，并从一个 ReLU 非线性产生最终张量。实现代码如下：

```
    #创建 MatMul 节点
    x_w = tf.matmul(X, W, name="MatMul")
    #创建 Add 节点
    x_w_b = tf.add(x_w, b, name="Add")
    #创建 ReLU 节点
    h = tf.nn.relu(x_w_b, name="ReLU")
```

在关闭之前，在该计算图上运行会话（使用由随机像素值生成的 100 张图像）并获取隐藏单元的输出 h，以下是完整的代码。

```
    import tensorflow as tf
    import numpy as np
    #创建输入占位符
    X = tf.placeholder(tf.float32, shape=[None, 784], name="X")
    weight_initer = tf.truncated_normal_initializer(mean=0.0, stddev=0.01)
    #创建网络参数
    W = tf.get_variable(name="Weight", dtype=tf.float32, shape=[784, 200],
initializer=weight_initer)
    bias_initer =tf.constant(0., shape=[200], dtype=tf.float32)
```

```
b = tf.get_variable(name="Bias", dtype=tf.float32, initializer=bias_initer)
#创建 MatMul 节点
x_w = tf.matmul(X, W, name="MatMul")
#创建 Add 节点
x_w_b = tf.add(x_w, b, name="Add")
#创建 ReLU 节点
h = tf.nn.relu(x_w_b, name="ReLU")
#添加一个操作来初始化全局变量
init_op = tf.global_variables_initializer()
#在会话中运行计算图
with tf.Session() as sess:
    #初始化变量
    sess.run(init_op)
    #创建词典
    d = {X: np.random.rand(100, 784)}
    #通过词典将值提供给占位符
    print(sess.run(h, feed_dict=d))
```

TensorFlow 计算图虽然功能强大，但可能会变得非常复杂。使用 TensorBoard 可视化图形可以帮助我们理解和调试计算图。为了让 TensorFlow 程序激活 TensorBoard，我们需要添加一些代码行。这会将 TensorFlow 操作导出到称为事件文件（或事件日志文件）的文件中。TensorBoard 能够读取此文件，并提供模型图形及其性能的一些可视化显示。下面编写一个简单的 TensorFlow 程序，并用 TensorBoard 可视化其计算图。

创建 a、b 两个常量并将它们加在一起，常量张量可以简单地由它们的值来定义。

```
import tensorflow as tf
#创建计算图
a = tf.constant(2)
b = tf.constant(3)
c = tf.add(a, b)
#在会话中运行计算图
with tf.Session() as sess:
    print(sess.run(c))
```

为了用 TensorBoard 把程序可视化，我们需要编写程序的日志文件。要编写事件日

志文件，先使用以下代码为这些日志创建一个写入器。

```
writer = tf.summary.FileWriter([logdir], [graph])
```

其中参数[logdir]为存储这些日志文件的文件夹，也可以选择[logdir]为'./graphs'等有意义的名称；参数[graph]为正在开发程序的图形，有以下两种方法可以获得图形。

- 使用 tf.get_default_graph()调用图形，该函数返回程序的默认图形。
- 将其设置为返回会话图形的 sess.graph（注意，这需要我们创建一个会话）。

在以下的示例中，可以看到这两种方法的应用。然而，第二种方法更常见。无论选用哪种方法，均要确保在定义图形后才创建一个写入器；否则，在 TensorBoard 上可视化的图形将不完整。将写入器添加到第一个示例中，并将图形可视化。

```
import tensorflow as tf
tf.reset_default_graph()   #清除先前单元格的已定义变量和操作
#创建图形
a = tf.constant(2)
b = tf.constant(3)
c = tf.add(a, b)
#在会话外创建写入器
#writer = tf.summary.FileWriter('./graphs', tf.get default graph())
#在会话中启动图形
with tf.Session() as sess:
    #或在会话中创建写入器
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    print(sess.run(c))
```

执行此代码，TensorFlow 将在当前目录中创建一个包含事件文件的目录。

在运行 Python 代码的目录下启动 TensorBoard 服务：

```
$tensorboard --logdir="./graphs" --port 6006
```

在浏览器中使用 http://<IP_Address>:6006/访问，该链接将引导我们进入 TensorBoard 页面，可视化示例代码生成的图形如图 7-8 所示。

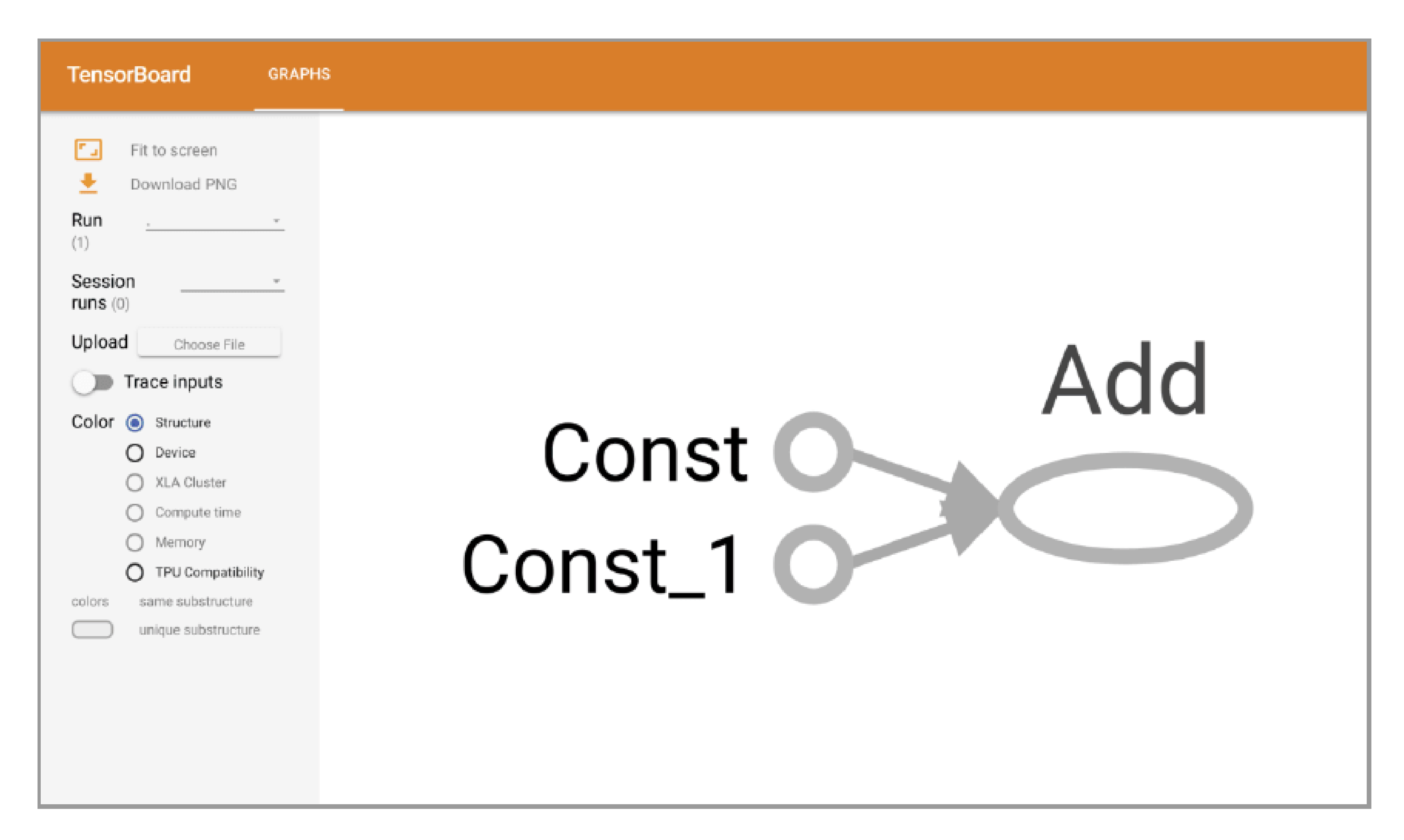


图 7-8　TensorBoard 页面中可视化示例代码生成的图形

图 7-9 中的图表显示了模型的各个部分。其中的节点“Const”和节点“Const_1”对应于代码中的 *a* 和 *b*，节点“Add”对应于 *c*。代码中给出的名称（*a*，*b* 和 *c*）只是 Python 名称，它们只在编写代码时帮助我们访问，名称对 TensorFlow 和 TensorBoard 没有任何意义。为了使 TensorBoard 了解我们的操作名称，必须明确地命名它们。

再次修改代码来添加名称。

```
import tensorflow as tf
tf.reset_default_graph()   #清除先前单元格的已定义变量和操作
#创建图形
a = tf.constant(2, name="a")
b = tf.constant(3, name="b")
c = tf.add(a, b, name="addition")
#在会话外创建写入器
#writer = tf.summary.FileWriter('./graphs', tf.get default graph())
#在会话中启动图形
with tf.Session() as sess:
    #或在会话中创建写入器
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    print(sess.run(c))
```

TensorBoard 页面中修改名称后生成的图形如图 7-9 所示。

图 7-9 TensorBoard 页面中修改名称后生成的图形

到目前为止，我们只关注了如何在 TensorBoard 中可视化图形。TensorBoard 还提供了其他类型（标量、图像和直方图）的可视化。在这一部分，我们将使用一种称为摘要（Summary）的特殊操作来将模型参数（如神经网络的权重和偏差）、度量（如损失或准确度值）和图像（如到网络的输入图像）可视化。

摘要是一个特殊的操作，它可接受常规张量并将汇总数据输出到磁盘（即事件文件）中。基本上，有以下 3 种主要类型的摘要。

- tf.summary.scalar()：用于写入单个标量值张量（如分类损失或准确度值）。
- tf.summary.histogram()：用于绘制非标量张量所有值的直方图（可用于可视化神经网络的权重或偏差矩阵）。
- tf.summary.image()：用于绘制图像（如网络的输入图像、自动编码器或 GAN 生成的输出图像）。

接下来，我们将更详细地讲解上述所有摘要类型。

tf.summary.scalar()用于编写随时间或迭代而变化的标量张量的值。在神经网络中，通常用于监视损失函数或分类精度的变化。下面让我们举一个简单的例子来理解这一点。

从标准正态分布 $N(0,1)$中随机挑选 100 个值，并依次绘制它们。一种方法是简单地创建一个变量，并从正态分布（平均值= 0 和标准差= 1）初始化变量，然后在会话中运行 for 循环并初始化。代码如下，编写摘要所需的步骤在代码中进行了说明。

```
    import tensorflow as tf
    tf.reset_default_graph()   #清除先前单元格的已定义变量和操作
    #创建标量变量
    x scalar = tf.get variable('x scalar', shape=[], initializer=tf.truncated
normal initializer(mean=0, stddev=1))
    #____步骤 1: ____创建标量摘要
    first summary=tf.summary.scalar(name='My first scalar summary', tensor=x
scalar)
    init = tf.global variables initializer()
    #在会话中启动图形
    with tf.Session() as sess:
       #____步骤 2: ____在会话中创建写入器
       writer = tf.summary.FileWriter('./graphs', sess.graph)
       for step in range(100):
          #循环变量的初始化
          sess.run(init)
          #____步骤 3: ____评估标量摘要
          summary = sess.run(first summary)
          #____步骤 4: ____将摘要添加到写入器（即事件文件）
          writer.add summary(summary, step)
       print('Done with writing the scalar summary')
```

如果我们希望观察值随时间或迭代的变化而变化，则直方图会派上用场。它用于绘制非标量张量值的直方图。在神经网络的情况下，它通常用于监测权重和偏差分布的变化，对检测网络参数的不规则非常有用（例如，当权重发生爆炸或异常收缩时）。

继续前面的示例，添加一个大小为 30×40 的矩阵，其条目来自标准正态分布。初始化该矩阵 100 次，并绘制其输入项随时间的分布。

```
    import tensorflow as tf
```

```
    tf.reset_default_graph()   #清除先前单元格的已定义变量和操作
    #创建变量
    x scalar=tf.get variable('x scalar', shape=[], initializer=tf.truncated
normal initializer(mean=0, stddev=1))
    x matrix=tf.get variable('x matrix',shape=[30,40], initializer=tf.truncated
normal initializer(mean=0, stddev=1))
    #____步骤 1: ____创建摘要
    #标量张量的标量摘要
    scalar summary = tf.summary.scalar('My scalar summary', x scalar)
    #非标量（即 2D 或矩阵）张量的直方图摘要
    histogram summary = tf.summary.histogram('My histogram summary', x matrix)

    init = tf.global variables initializer()

    #在会话中启动图形
    with tf.Session() as sess:
        #____步骤 2: ____在会话中创建写入器
        writer = tf.summary.FileWriter('./graphs', sess.graph)
        for step in range(100):
            #循环变量的几个初始化操作
            sess.run(init)
            #____步骤 3: ____评估合并的摘要
            summary1, summary2 = sess.run([scalar summary, histogram summary])
            #____步骤 4: ____将摘要添加到写入器（即事件文件）以写入硬盘
            writer.add summary(summary1, step)
            #重复直方图摘要的步骤 4
            writer.add summary(summary2, step)
        print('Done writing the summaries')
```

在 TensorBoard 中，顶部菜单中添加了“分布”和“直方图”两个选项卡。“分布”选项卡包含一个图表，显示通过步骤（x 轴）、张量值（y 轴）的分布。

我们需要运行每个摘要（例如 sess.run([scalar_summary,histogram_summary])），然后使用写入器将它们中的每一个写入磁盘。实际上，我们可以使用任意数量的摘要来跟踪模型中的不同参数，这使得运行和写入摘要极其低效。解决方法是通过 tf.summary.

merge_all()合并图表中的所有摘要，并在会话中立即运行它们。代码更改如下：

```
    import tensorflow as tf
    tf.reset_default_graph()   #清除先前单元格的已定义变量和操作
    #创建变量
    x scalar=tf.get variable('x scalar',shape=[],initializer=tf.truncated
normal initializer(mean=0, stddev=1))
    x matrix=tf.get variable('x matrix',shape=[30,40],initializer=tf.truncat
ed normal initializer(mean=0, stddev=1))
    #____步骤 1: ____创建摘要
    #标量张量的标量摘要
    scalar summary = tf.summary.scalar('My scalar summary', x scalar)
    #非标量（即 2D 或矩阵）张量的直方图摘要
    histogram summary=tf.summary.histogram('My histogram summary', x matrix)
    #____步骤 2: ____合并所有摘要
    merged = tf.summary.merge all()
    init = tf.global variables initializer()
    #在会话中启动图形
    with tf.Session() as sess:
        #____步骤 3: ____在会话中创建写入器
        writer = tf.summary.FileWriter('./graphs', sess.graph)
        for step in range(100):
            #循环变量的几个初始化操作
            sess.run(init)
            #____步骤 4: ____评估合并的摘要
            summary = sess.run(merged)
            #____步骤 5: ____将摘要添加到写入器（即事件文件）以写入硬盘
            writer.add summary(summary, step)
        print('Done writing the summaries')
```

tf.summary.image()用于写出和可视化张量作为图像。在神经网络的情况下，它通常用于追踪馈送到网络（如在每批中）或在输出中生成的图像（如在自动编码器中重建的图像）。一般来说，它可以用于绘制任何张量。例如，我们可以将大小为 30×40 的权重矩阵可视化为 30 像素×40 像素的图像。

图像摘要可以使用以下代码创建。

```
tf.summary.image(name, tensor, max_outputs=3)
```

其中 name 为生成节点的名称（即操作）；tensor 为要写成图像摘要的期望张量；max_outputs 为张量中生成图像的最大元素数量。但这是什么意思呢？答案是，在于张量的形状。

我们提供给 tf.summary.image()的张量必须是形状的四维张量[batch_size、height、width、channels]，其中 batch_size 为批次中图像的数量；height 和 width 分别为高度和宽度，它们决定图像的大小；channels 为通道，它的值包括 1 用于灰度图像、3 用于 RGB（即彩色）图像、4 用于 RGBA 图像（A 代表 Alpha 值）。

以下是一个非常简单的例子。

```
import tensorflow as tf
tf.reset_default_graph()   #清除先前单元格的已定义变量和操作

#创建变量
w gs = tf.get variable('W Grayscale', shape=[30, 10], initializer=tf.truncated
normal initializer(mean=0, stddev=1))
w c = tf.get variable('W Color', shape=[50, 30], initializer=tf.truncated
normal initializer(mean=0, stddev=1))

#____步骤 0: ____将变量重新塑造成四维张量
w gs reshaped = tf.reshape(w gs, (3, 10, 10, 1))
w c reshaped = tf.reshape(w c, (5, 10, 10, 3))
#____步骤 1: ____创建摘要
gs summary = tf.summary.image('Grayscale', w gs reshaped)
c summary = tf.summary.image('Color', w c reshaped, max outputs=5)
#____步骤 2: ____合并所有摘要
merged = tf.summary.merge all()
#创建用于初始化所有变量的操作
init = tf.global variables initializer()
#在会话中启动图形
with tf.Session() as sess:
    #____步骤 3: ____在会话中创建写入器
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    #初始化所有变量
```

```
    sess.run(init)
    #____步骤 4: ____评估合并的操作以获得摘要
    summary = sess.run(merged)
    #____步骤 5: ____将摘要添加到写入器（即事件文件）以写入硬盘
    writer.add summary(summary)
    print('Done writing the summaries')
```

打开 TensorBoard 界面并切换到 IMAGES 选项卡，图像应该类似于图 7-10。

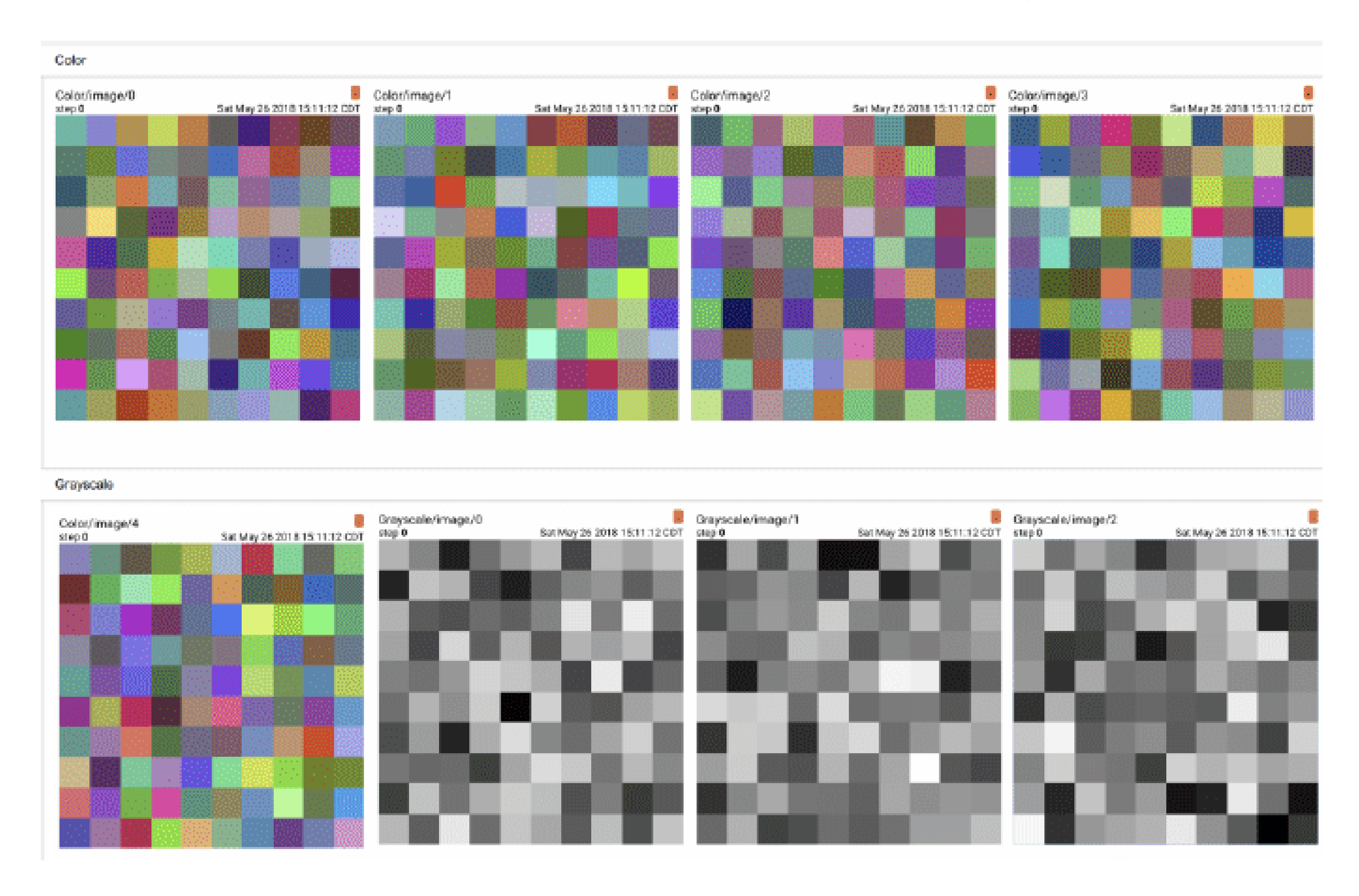


图 7-10　在 TensorBoard 中生成的图像

我们可以将任何尺寸的其他图像添加到摘要中，并将它们绘制在 TensorBoard 中。

下面讨论如何将参数保存到磁盘并从磁盘恢复已保存的参数。网络的可保存/可恢复的参数是变量（即权重和偏差）。为了保存和恢复变量，你所需要做的就是在图结尾调用 tf.train.Saver()。

```
#创建图
X = tf.placeholder(..)
Y = tf.placeholder(..)
w = tf.get variable(..)
b = tf.get variable(..)
...
loss = tf.losses.mean squared error(..)
optimizer = tf.train.AdamOptimizer(..).minimize(loss)
...
```

```
saver = tf.train.Saver()
```

在训练模式下，可以在会话中初始化变量并运行网络。在训练结束时，可以使用saver.save()保存变量。

```
with tf.Session() as sess:
    sess.run(tf.global variables initializer())
    #训练模型
    for step in range(steps):
        sess.run(optimizer)
        ...
    saved_path = saver.save(sess, './my-model', global_step=step)
```

上述代码将创建3个文件（data、index、meta），并且带有保存模型的步骤后缀。

在测试模式下，可以在会话中使用saver.restore()恢复变量并验证或测试我们的模型。

```
#测试
with tf.Session() as sess:
    saver.restore(sess, './my-model')
    ...
```

为了实现在TensorFlow中保存和恢复两个变量，我们将创建一个包含两个变量的图。创建a = [3 3]和b = [5 5 5] 两个变量，实现代码如下：

```
import tensorflow as tf
#创建变量a和b
a = tf.get_variable("A", initializer=tf.constant(3, shape=[2]))
b = tf.get_variable("B", initializer=tf.constant(5, shape=[3]))
```

注意，上述代码中小写字母*a*、*b*为Python名称，大写字母A、B为TensorFlow名称。当我们想要导入图来恢复数据时，这一点很重要。

变量在使用前需要初始化。初始化所有变量的实现代码如下：

```
#初始化所有变量
init_op = tf.global_variables_initializer()
```

在会话中，初始化变量并运行以查看值。

```
#运行会话
with tf.Session() as sess:
```

```
    #初始化会话中的所有变量
    sess.run(init_op)
    #运行会话以获取变量的值
    a_out, b_out = sess.run([a, b])
    print('a = ', a_out)
    print('b = ', b_out)
```

输出结果：

```
a =  [3 3]
b =  [5 5 5]
```

所有变量都存在于会话范围内。会话结束后，我们将放弃这些变量。为了保存变量，我们将在图中使用 tf.train.Saver()调用保存函数。该函数将查找图中的所有变量，我们可以在_var_list 中看到所有变量的列表。创建一个 saver 对象并查看对象中的_var_list，实现代码如下：

```
#创建 saver 对象
saver = tf.train.Saver()
for i, var in enumerate(saver._var_list):
    print('Var {}: {}'.format(i, var))
```

输出结果：

```
Var 0: <tf.Variable 'A:0' shape=(2,) dtype=int32 ref>
Var 1: <tf.Variable 'B:0' shape=(3,) dtype=int32_ref>
```

可见，我们的图由上面列出的两个变量组成。注意，变量名称末尾有 0。

既然保存对象是在图形中创建的，那么在会话中，我们可以调用函数 saver.save()将变量保存在磁盘中。我们必须将创建的会话（sess）和想要保存变量的文件路径传递给函数 save()。

```
#运行会话
with tf.Session() as sess:
    #初始化会话中的所有变量
    sess.run(init op)
    #将变量保存在磁盘中
    saved_path = saver.save(sess, './saved_variable')
```

```
    print('model saved in {}'.format(saved_path))
```

模型保存在./saved_variable 中。如果检查工作目录，会注意到创建了 3 个名为 saved_variable 的新文件。

```
import os
for file in os.listdir('.'):
    if 'saved_variable' in file:
        print(file)
```

输出结果：

```
saved_variable.data-00000-of-00001
saved variable.meta
saved_variable.index
```

这里的.data 文件包含变量值；.meta 文件包含图形结构；.index 文件标识检查点。

使用 saver.restore()在会话中加载已保存的变量。

```
#运行会话
with tf.Session() as sess:
    #恢复保存的变量
    saver.restore(sess, './saved variable')
    #打印加载的变量
    a out, b out = sess.run([a, b])
    print('a = ', a out)
    print('b = ', b_out)
```

输出结果：

```
INFO:tensorflow:Restoring parameters from ./saved variable
a =  [3 3]
b =  [5 5 5]
```

注意，这次没有在会话中初始化变量，而是从磁盘中恢复它们。为了恢复参数，应该定义图形。由于在顶部定义了图形，因此恢复参数时没有出现问题。

除了在代码中定义图形，还可以从.meta 文件恢复图形。当保存这些变量时，会创建一个包含图形结构的.meta 文件。因此，我们可以使用 tf.train.import_ meta_graph()导入元图并恢复图的值。导入图形并查看图中的所有张量，实现代码如下：

```
#删除当前图形
tf.reset default graph()
#从文件中导入图形
imported graph = tf.train.import meta graph('saved variable.meta')
#列出图中的所有张量
for tensor in tf.get default graph().get operations():
    print (tensor.name)
```

现在有导入的图形，如果对张量 A 和 B 感兴趣，可以用以下命令恢复参数。

```
#运行会话
with tf.Session() as sess:
    #恢复保存的变量
    imported graph.restore(sess, './saved variable')
    #打印加载的变量
    a out, b out = sess.run(['A:0','B:0'])
    print('a = ', a out)
    print('b = ', b_out)
```

输出结果：

```
INFO:tensorflow:Restoring parameters from ./saved variable
a =  [3 3]
b =  [5 5 5]
```

注意，在 sess.run()中，我们是使用张量'A:0'和'B:0'的 TensorFlow 名称，而不是使用 a 和 b。

Saver 主要用于生成变量的检查点。SavedModel 将取代现有的 TensorFlow 推理模型格式，作为导出 TensorFlow 图形进行服务的标准方式。TensorFlow 的 SavedModel 格式包括有关模型的所有信息（图形、检查点状态、其他元数据）。所以如果想在 Java 中使用它，可以使用 SavedModelBundle.load()。

Python 中的保存模型代码如下：

```
import tensorflow as tf
import os
x = tf.placeholder(tf.float32, name='x')      #模型输入
w  = tf.get variable('w',shape=[1,1], initializer=tf.random normal
initializer())
```

```
    b = tf.get variable('b', shape=[1, 1], initializer=tf.zeros initializer())
    y = tf.add(b, tf.matmul(x, w), name='y')      #模型输出
    training steps = 51
    with tf.Session() as sess:
       sess.run(tf.global variables initializer())
       for train step in range(1, training steps):
          #...
          #运行训练操作
          #...
          if train_step % 50 == 0:                  #每隔 50 步保存一次
             #将 train_step 添加到 export_dir
             export dir = os.getcwd() + '/savedModel-' + str(train step)
             inputs_dict = {'x_in': x}              #字符串名称不等于张量名称
             outputs_dict = {'y': y}                #字符串名称等于张量名称
             tf.saved model.simple save(sess, export dir, inputs=inputs dict,
outputs=outputs dict)
             print('y: %.4f' % (sess.run(y, feed dict={ x: [[1]] })))

    tf.reset_default_graph()     #清除图结构
    with tf.Session() as sess:
       #加载图形和变量
       export dir = os.getcwd() + '/savedModel-50'
       meta graph def = tf.saved model.loader.load(sess, [tf.saved model.tag
constants.SERVING], export dir)
       #获取将输入/输出字符串名称映射到张量的签名定义
       sig def = meta graph def.signature def[tf.saved model.signature constants.
DEFAULT SERVING SIGNATURE DEF KEY]
       x_name = sig_def.inputs['x_in'].name              #获取输入张量名称
       x = tf.get default graph().get tensor by name(x name)
       y_name = sig_def.outputs['y'].name                #获取输出张量名称
       y = tf.get default graph().get tensor by name(y name)
       print('y: %.4f' % (sess.run(y, feed dict={ x: [[1]] })))
    #Output:
    #x: -0.2909
    #x: -0.2909
```

训练时执行的计算过程和推理时执行的计算过程不一样。

神经网络是深度学习的第一步。深度学习的名字来源于计算机科学家希望用神经元的相同功能来模拟大脑结构的概念。深度学习的关键点是，它可以分离不能线性分离的数据。

要构建任何分类器，代码需要如下部分。

①为网络准备所需的库，输入数据和超参数。

②建立网络图。

③训练网络。

④测试网络。

为了能够使用 matplotlib，先安装 matplotlib 依赖的 tkinter。

```
$sudo apt-get install python3-tk
```

从导入所需的库开始。

```
#imports
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

在这里，我们使用 MNIST 数据集。MNIST 是手写数字的数据集，是深度学习数据集的基准。使用 MNIST 的另一个原因是通过 TensorFlow 很容易访问。

该数据集包含 55 000 个训练示例，其中 5 000 个用于验证的示例，10 000 个用于测试的示例。这些数字图像已经进行了尺寸标准化并以固定尺寸图像（28 像素×28 像素）为中心。图像中的像素点用值为 0～1 的数表示。为了简单起见，每幅图像都被平展并转换为 784 个特征的一维 numpy 阵列（28×28）。

我们可以轻松导入数据集并查看训练、测试和验证集的大小。

```
#导入 MNIST 数据
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
print("Size of:")
print("- Training-set:\t\t{}".format(len(mnist.train.labels)))
print("- Test-set:\t\t{}".format(len(mnist.test.labels)))
```

```
print("- Validation-set:\t{}".format(len(mnist.validation.labels)))
```

输出结果：

```
Size of:
- Training-set:         55000
- Test-set:             10000
- Validation-set:       5000
```

超参数是使网络无法学习的重要参数。我们必须在外部指定超参数，实现代码如下：

```
#超参数
learning_rate = 0.001  #优化学习率
epochs = 10  #训练回合数
batch_size = 100  #训练批次大小
display_freq = 100  #显示训练结果的频率
#网络参数
#MNIST 图像在每个维度上是 28 个像素
img_h = img_w = 28
#图像存储在这个长度的一维数组中
img_size_flat = img_h * img_w
#类的数量
n_classes = 10
#第一个隐藏层中的单元数量
h1 = 200
```

在开始构建图之前，我们需要快速地完成一些功能。因此，我们不会多次调用它们，而是定义一些有用的函数，在图中调用这些函数。最重要的是用于创建权重和偏差变量的函数。由于正在创建一个神经网络，因此我们需要一个完全连接的层来将上一层的所有节点连接到我们的层。

```
#权重和偏差包装器，可以用于设置卷积核的权重
def weight_variable(name, shape):
    """
    用适当的初始化创建一个权重变量
    name: 权重变量名称
```

```
    shape: 权重变量形状
    return: 初始化的权重变量
    """
    initer = tf.truncated_normal_initializer(stddev=0.01)
    return tf.get_variable('W_' + name,
                           dtype=tf.float32,
                           shape=shape,
                           initializer=initer)
def bias_variable(name, shape):
    """
    用适当的初始化创建一个偏差变量
    name: 偏差变量名称
    shape: 偏置变量形状
    return: 初始化的偏差变量
    """
    initial = tf.constant(0., shape=shape, dtype=tf.float32)
    return tf.get_variable('b_' + name,
                           dtype=tf.float32,
                           initializer=initial)
def fc_layer(x, num_nodes, name, use_relu=True):
    """
    创建一个完全连接的图层
    :param x: 来自上一层的输入
    :param num_nodes: 完全连接层中隐藏单元的数量
    :param name: 图层名称
    :param use_relu:用来决定是否添加ReLU非线性到图层的布尔值
    :return: 输出数组
    """
    in_dim = x.get_shape()[1]
    W = weight_variable(name, shape=[in_dim, num_nodes])
    b = bias_variable(name, [num_nodes])
    layer = tf.matmul(x, W)
    layer += b
    if use_relu:
        layer = tf.nn.relu(layer)
```

```
        return layer
```

有了帮助函数，我们可以创建自己的图结构，实现代码如下：

```
    #创建图结构
    #inputs(x)和 outputs(y)的占位符
    x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name='X')
    y = tf.placeholder(tf.float32, shape=[None, n_classes], name='Y')
    fc1 = fc_layer(x, h1, 'FC1', use_relu=True)
    output_logits = fc_layer(fc1, n_classes, 'OUT', use_relu=False)
    #定义损失函数、优化器和准确度
    loss=tf.reduce mean(tf.nn.softmax cross entropy with logits(labels=y,
logits=output_logits), name='loss')
    optimizer=tf.train.AdamOptimizer(learning rate=learning rate,name= 'Adam-
op').minimize(loss)
    correct prediction = tf.equal(tf.argmax(output logits, 1), tf.argmax(y, 1),
name='correct_pred')
    accuracy=tf.reduce_mean(tf.cast(correct_prediction,tf.float32),name='accuracy')
    #网络预测
    cls_prediction = tf.argmax(output_logits, axis=1, name='predictions')
    #初始化变量
    init = tf.global_variables_initializer()
```

创建出图结构以后，我们就可以在会话上运行它（运行时可以使用 tf.Session()）。注意，一旦运行单元，会话就会结束，并将丢失所有信息。因此，我们将定义一个 InteractiveSession 来保存参数用于测试。

```
    #在会话中启动图形
    sess = tf.InteractiveSession()#是使用 InteractiveSession,而不是使用 Session
来测试单独单元中的网络
    sess.run(init)
    #每个回合的训练迭代次数
    num_tr_iter = int(mnist.train.num_examples / batch_size)
    for epoch in range(epochs):
        print('Training epoch: {}'.format(epoch+1))
        for iteration in range(num_tr_iter):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
```

```
            #运行优化操作（反向传播）
            feed_dict_batch = {x: batch_x, y: batch_y}
            sess.run(optimizer, feed_dict=feed_dict_batch)
            if iteration % display_freq == 0:
                #计算并显示批损失和准确度
                loss_batch, acc_batch = sess.run([loss, accuracy],
                                                 feed_dict=feed_dict_batch)
                print("iter {0:3d}:\t Loss={1:.2f},\tTraining Accuracy={2:.01%}".
                      format(iteration, loss_batch, acc_batch))
        #在每个回合后运行验证
        feed_dict_valid = {x: mnist.validation.images, y: mnist.validation.labels}
        loss_valid, acc_valid = sess.run([loss, accuracy], feed_dict=feed_dict_
valid)
        print('---------------------------------------------------------')
        print("Epoch: {0}, validation loss: {1:.2f}, validation accuracy:
{2:.01%}". format(epoch + 1, loss_valid, acc_valid))
        print('---------------------------------------------------------')
```

使用上述代码训练好模型后，现在是测试模型的时候了。我们将定义一些辅助函数来绘制一些图像及其相应的预测和真实类，还将可视化一些错误分类的样本，以了解为什么神经网络未能正确分类。

```
    def plot_images(images, cls_true, cls_pred=None, title=None):
        """
        用 3×3 子图创建图
        :param images: 要绘制的图像数组(9, img_h*img_w)
        :param cls_true: 相应的真值标签(9,)
        :param cls_pred: 相应的真值标签(9,)
        """
        fig, axes = plt.subplots(3, 3, figsize=(9, 9))
        fig.subplots_adjust(hspace=0.3, wspace=0.3)
        img_h = img_w = np.sqrt(images.shape[-1]).astype(int)
        for i, ax in enumerate(axes.flat):
            #绘制图像
            ax.imshow(images[i].reshape((img_h, img_w)), cmap='binary')
```

```
        #显示实际和预测的类
        if cls_pred is None:
            ax_title = "True: {0}".format(cls_true[i])
        else:
            ax_title="True: {0}, Pred: {1}".format(cls_true[i], cls_pred[i])
        ax.set_title(ax_title)
        #从图中删除刻度线
        ax.set_xticks([])
        ax.set_yticks([])
    if title:
        plt.suptitle(title, size=20)
    plt.show()
def plot_example_errors(images, cls_true, cls_pred, title=None):
    """
    绘制错误分类图像示例的函数
    :param images: 所有图像的数组, (#imgs, img_h*img_w)
    :param cls_true: 相应的真实值标签, (#imgs,)
    :param cls_pred: 相应的预测标签, (#imgs,)
    """
    #否定布尔数组
    incorrect = np.logical_not(np.equal(cls_pred, cls_true))
    #从测试集中获取错误分类的图像
    incorrect_images = images[incorrect]
    #获取这些图像的真实和预测的类
    cls_pred = cls_pred[incorrect]
    cls_true = cls_true[incorrect]
    #绘制前 9 张图像
    plot_images(images=incorrect_images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9],
                title=title)
```

服务器版本的 Ubuntu 需要安装图形界面：

```
$sudo apt install lubuntu-desktop
```

然后重启：

```
$reboot
```

运行模型：

```
#训练后测试网络
#测试模型的准确度
feed_dict_test = {x: mnist.test.images, y: mnist.test.labels}
loss_test, acc_test = sess.run([loss, accuracy], feed_dict=feed_dict_test)
print('---------------------------------------------------------')
print("Test loss: {0:.2f}, test accuracy: {1:.01%}".format(loss_test, acc_test))
print('---------------------------------------------------------')
#绘制正确和错误的分类例子
cls_pred = sess.run(cls_prediction, feed_dict=feed_dict_test)
cls_true = np.argmax(mnist.test.labels, axis=1)
plot_images(mnist.test.images, cls_true, cls_pred, title='Correct Examples')
plot_example_errors(mnist.test.images,cls_true,cls_pred,title='Misclassified Examples')
```

使用卷积神经网络可以减少分类错误。接下来，我们在 TensorFlow 中实现一个简单的卷积神经网络——用两个卷积层后接两个全连接层。

加载 MNIST 数据的辅助函数：

```
def load_data(mode='train'):
    """
    下载和加载 MNIST 数据的函数
    :param mode: train 或 test
    :return: 图像和相应的标签
    """
    mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
    if mode == 'train':
        x_train, y_train, x_valid, y_valid = mnist.train.images, mnist.train.labels, \ mnist.validation.images, mnist. validation.labels
        x_train, _ = reformat(x_train, y_train)
```

```
            x_valid, _ = reformat(x_valid, y_valid)
            return x_train, y_train, x_valid, y_valid
        elif mode == 'test':
            x_test, y_test = mnist.test.images, mnist.test.labels
            x_test, _ = reformat(x_test, y_test)
        return x_test, y_test
    def reformat(x, y):
        """
        将数据重新格式化为卷积层可接受的格式
        :param x: 输入数组
        :param y: 相应的标签
        :return: 重新塑造的输入和标签
        """
        img_size, num_ch, num_class = int(np.sqrt(x.shape[-1])), 1, len(np.unique
(np.argmax(y, 1)))
        dataset = x.reshape((-1, img_size, img_size, num_ch)).astype(np.float32)
        labels = (np.arange(num_class) == y[:, None]).astype(np.float32)
        return dataset, labels
    def randomize(x, y):
        """随机化数据样本及其相应标签的顺序"""
        permutation = np.random.permutation(y.shape[0])
        shuffled_x = x[permutation, :, :, :]
        shuffled_y = y[permutation]
        return shuffled_x, shuffled_y
    def get_next_batch(x, y, start, end):
        x_batch = x[start:end]
        y_batch = y[start:end]
        return x_batch, y_batch
```

在训练模式下使用已定义的辅助函数加载训练和验证图像及相应的标签，并显示数据集的大小。

```
    x_train, y_train, x_valid, y_valid = load_data(mode='train')
    print("Size of:")
    print("- Training-set:\t\t{}".format(len(y_train)))
    print("- Validation-set:\t{}".format(len(y_valid)))
```

输出结果：

```
Extracting MNIST data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Size of:
- Training-set:      55000
- Validation-set:    5000
```

超参数：

```
logs_path = "./logs"      #指向我们想要为 TensorBoard 保存日志的路径
lr = 0.001                #优化的初始学习率
epochs = 10               #训练回合的总数
batch_size = 100          #训练批大小
display_freq = 100        #显示训练结果的频率
```

网络配置：

```
#第一个卷积层
filter_size1 = 5          #卷积核为 5 像素×5 像素
num_filters1 = 16         #有 16 个这样的卷积核
stride1 = 1               #滑动窗口的步幅
#第二个卷积层
filter_size2 = 5          #卷积核为 5 像素×5 像素
num_filters2 = 32         #有 32 个这样的卷积核
stride2 = 1               #滑动窗口的步幅
#全连接层
h1 = 128                  #完全连接层中的神经元数量
```

创建网络辅助函数。

用于创建新变量的辅助函数：

```
#权重和偏差包装器
def weight_variable(shape):
    """
    用适当的初始化创建一个权重变量
```

```
    :param name: 权重名称
    :param shape: 权重形状
    :return: 初始化后的权重变量
    """
    initer = tf.truncated_normal_initializer(stddev=0.01)
    return tf.get_variable('W',
                           dtype=tf.float32,
                           shape=shape,
                           initializer=initer)
def bias_variable(shape):
    """
    用适当的初始化器创建一个偏差变量
    :param name: 偏差变量名称
    :param shape: 偏差变量形状
    :return: 初始化后的偏差变量
    """
    initial = tf.constant(0., shape=shape, dtype=tf.float32)
    return tf.get_variable('b', dtype=tf.float32, initializer=initial)
```

用于创建新卷积层的帮助函数：

```
def conv_layer(x, filter_size, num_filters, stride, name):
    """
    创建一个2D卷积层
    :param x: 来自上一层的输入
    :param filter_size: 每个过滤器的大小
    :param num_filters: 过滤器的数量
    :param stride: 过滤器跨度
    :param name: 图层名称
    :return: 输出数组
    """
    with tf.variable_scope(name):
        num_in_channel = x.get_shape().as_list()[-1]
        shape = [filter_size, filter_size, num_in_channel, num_filters]
        W = weight_variable(shape=shape)
        tf.summary.histogram('weight', W)
```

```
        b = bias_variable(shape=[num_filters])
        tf.summary.histogram('bias', b)
        layer = tf.nn.conv2d(x, W, strides=[1, stride, stride, 1],
                             padding="SAME")
        layer += b
        return tf.nn.relu(layer)
```

用于创建一个新的、最大池图层的帮助函数：

```
def max_pool(x, ksize, stride, name):
    """
    创建一个最大池化层
    :param x: 最大池化层的输入
    :param ksize: 最大池化层过滤器的大小
    :param stride: 最大池卷积核的步幅
    :param name: 图层名称
    :return: 输出数组
    """
    return tf.nn.max_pool(x,
                          ksize=[1, ksize, ksize, 1],
                          strides=[1, stride, stride, 1],
                          padding="SAME",
                          name=name)
```

用于展开图层的辅助函数：

```
def flatten_layer(layer):
    """
    展平卷积层的输出,使其馈送到全连接层
    :param layer: 输入数组
    :return: 展平后的数组
    """
    with tf.variable_scope('Flatten_layer'):
        layer_shape = layer.get_shape()
        num_features = layer_shape[1:4].num_elements()
        layer_flat = tf.reshape(layer, [-1, num_features])
    return layer_flat
```

用于创建新全连接层的辅助函数：

```
def fc layer(x, num units, name, use relu=True):
    """
    创建一个全连接层
    :param x: 来自上一层的输入
    :param num_units: 全连接层中隐藏单元的数量
    :param name: 图层名称
    :param use_relu: 用来决定是否添加 ReLU 非线性到图层的布尔值
    :return: 输出数组
    """
    with tf.variable_scope(name):
        in_dim = x.get_shape()[1]
        W = weight_variable(shape=[in_dim, num_units])
        tf.summary.histogram('weight', W)
        b = bias_variable(shape=[num_units])
        tf.summary.histogram('bias', b)
        layer = tf.matmul(x, W)
        layer += b
        if use_relu:
            layer = tf.nn.relu(layer)
        return layer
```

创建卷积神经网络图。

输入 x 和相应标签 y 的占位符：

```
#Scope 可以帮助你在使用 TensorBoard 时分解模型
with tf.name_scope('Input'):
    x = tf.placeholder(tf.float32, shape=[None, img h, img w, n channels],
name='X')
    y = tf.placeholder(tf.float32, shape=[None, n_classes], name='Y')
```

创建网络层：

```
conv1 = conv layer(x, filter size1, num filters1, stride1, name='conv1')
pool1 = max_pool(conv1, ksize=2, stride=2, name='pool1')
conv2 = conv_layer(pool1, filter_size2, num_filters2, stride2, name='conv2')
```

```
    pool2 = max_pool(conv2, ksize=2, stride=2, name='pool2')
    layer_flat = flatten_layer(pool2)
    fc1 = fc_layer(layer_flat, h1, 'FC1', use_relu=True)
    output_logits = fc_layer(fc1, n_classes, 'OUT', use_relu=False)
```

定义损失函数、优化器、准确度和预测类：

```
    with tf.variable_scope('Train'):
        with tf.variable_scope('Loss'):
            loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y,
logits=output_logits), name='loss')
        tf.summary.scalar('loss', loss)
        with tf.variable_scope('Optimizer'):
            optimizer = tf.train.AdamOptimizer(learning_rate=lr, name='Adam-
op').minimize(loss)
        with tf.variable_scope('Accuracy'):
            correct_prediction = tf.equal(tf.argmax(output_logits, 1), tf.argmax(y,
1), name='correct_pred')
            accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32),
name='accuracy')
        tf.summary.scalar('accuracy', accuracy)
        with tf.variable_scope('Prediction'):
            cls_prediction = tf.argmax(output_logits, axis=1, name='predictions')
```

初始化变量，并合并所有摘要：

```
    #初始化变量
    init = tf.global_variables_initializer()
    #合并所有摘要
    merged = tf.summary.merge_all()
```

训练：

```
    sess = tf.InteractiveSession()
    sess.run(init)
    global_step = 0
    summary_writer = tf.summary.FileWriter(logs_path, sess.graph)
    #每个回合的训练迭代次数
    num_tr_iter = int(len(y_train) / batch_size)
```

```
    for epoch in range(epochs):
        print('Training epoch: {}'.format(epoch + 1))
        x_train, y_train = randomize(x_train, y_train)
        for iteration in range(num_tr_iter):
            global_step += 1
            start = iteration * batch_size
            end = (iteration + 1) * batch_size
            x_batch, y_batch = get_next_batch(x_train, y_train, start, end)
            #运行优化操作（反向传播）
            feed_dict_batch = {x: x_batch, y: y_batch}
            sess.run(optimizer, feed_dict=feed_dict_batch)
            if iteration % display_freq == 0:
                #计算并显示批损失和准确度
                loss_batch, acc_batch, summary_tr = sess.run([loss, accuracy,
merged], feed_dict=feed_dict_batch)
                summary_writer.add_summary(summary_tr, global_step)
                print("iter {0:3d}:\t Loss={1:.2f},\tTraining Accuracy={2:.01%}".
                      format(iteration, loss_batch, acc_batch))
        #在每个回合后运行验证
        feed_dict_valid = {x: x_valid, y: y_valid}
        loss_valid, acc_valid = sess.run([loss, accuracy], feed_dict=feed_dict_valid)
        print('---------------------------------------------------------')
        print("Epoch: {0}, validation loss: {1:.2f}, validation accuracy:
{2:.01%}".format(epoch+1, loss_valid, acc_valid))
        print('---------------------------------------------------------')
```

测试：

```
    #完成训练后测试网络
    x_test, y_test = load_data(mode='test')
    feed_dict_test = {x: x_test, y: y_test}
    loss_test, acc_test = sess.run([loss, accuracy], feed_dict=feed_dict_test)
    print('---------------------------------------------------------')
    print("Test loss: {0:.2f}, test accuracy: {1:.01%}".format(loss_test,
acc_test))
```

```
    print('--------------------------------------------------------')
    #绘制一些正确和错误分类的例子
    cls_pred = sess.run(cls_prediction, feed_dict=feed_dict_test)
    cls_true = np.argmax(y_test, axis=1)
    plot_images(x_test, cls_true, cls_pred, title='Correct Examples')
    plot_example_errors(x_test,  cls_true,  cls_pred,  title='Misclassified
Examples')
    plt.show()
```

用 TensorFlow 读入一个图像列表：

```
    import tensorflow as tf
    filenames = ['/home/aaa/firefox.jpg']
    filename_queue = tf.train.string_input_producer(filenames)
    reader = tf.WholeFileReader()
    key, value = reader.read(filename_queue)
    images = tf.image.decode_jpeg(value, channels=3)
```

TensorFlow 提供了更高级别的 Estimator API，其中包含用于训练和预测数据的预建模型。TensorFlow 官方模型（https://github.com/tensorflow/models/tree/master/official）是使用 TensorFlow 高级 API 的示例模型集合。

训练并保存模型：

```
    $python3 mnist.py --export_dir /home/ai/mnist_saved_model
```

或者使用 nohup 命令脱离控制台运行：

```
    $nohup python3 mnist.py --export_dir /home/ai/mnist_saved_model &
```

显示模型文件：

```
    $saved_model_cli show --dir /home/ai/mnist_saved_model/1530443317 --all
```

使用 TensorFlow 中的方法加载图像：

```
    from tensorflow.python.lib.io import file_io
    d = np.load(file_io.FileIO("F:/models-master/official/mnist/examples.npy",
mode='rb'))
    plt.imshow(d[1], cmap=plt.cm.gray)
    plt.show()
```

运行模型文件：

```
$saved_model_cli run --dir /home/ai/mnist_saved_model/1530443317 --tag_set
serve --signature_def classify --inputs image=examples.npy
```

可能的输出结果如下：

```
Result for output key classes:
[5 3]
Result for output key probabilities:
[[  1.53558474e-07   1.95694142e-13   1.31193523e-09   5.47467265e-03
    5.85711526e-22   9.94520664e-01   3.48423509e-06   2.65365645e-17
    9.78631419e-07   3.15522470e-08]
 [  1.22413359e-04   5.87615965e-08   1.72251271e-06   9.39960718e-01
    3.30306928e-11   2.87386645e-02   2.82353517e-02   8.21146413e-18
    2.52568233e-03   4.15460236e-04]]
```

7.3.7 一维卷积

要手动计算一维卷积，可以在输入上滑动内核，求逐元素乘法并对它们求和。最简单的方法是 padding=0, stride=1。因此，如果你的输入为[1,0,2,3,0,1,1]，并且卷积核为[2,1,3]，则卷积的结果为[8,11,7,9,4]，这是按以下方式计算的。

$8 = 1 \times 2 + 0 \times 1 + 2 \times 3$

$11 = 0 \times 2 + 2 \times 1 + 3 \times 3$

$7 = 2 \times 2 + 3 \times 1 + 0 \times 3$

$9 = 3 \times 2 + 0 \times 1 + 1 \times 3$

$4 = 0 \times 2 + 1 \times 1 + 1 \times 3$

TensorFlow 的函数 tf.nnconv1d()分批计算卷积，为了在 TensorFlow 中执行此操作，我们需要以正确的格式提供数据。处理批次的操作，假设 Tensor 的第一个维度是批量维度，这里设置批大小为 1。计算一维卷积的代码如下：

```
import tensorflow as tf
i = tf.constant([1, 0, 2, 3, 0, 1, 1], dtype=tf.float32, name='i')
```

```
k = tf.constant([2, 1, 3], dtype=tf.float32, name='k')
print (i, '\n', k, '\n')
data   = tf.reshape(i, [1, int(i.shape[0]), 1], name='data')
kernel = tf.reshape(k, [int(k.shape[0]), 1, 1], name='kernel')
print (data, '\n', kernel, '\n')
stride=1;
res = tf.squeeze(tf.nn.conv1d(data, kernel, stride, 'VALID'))
with tf.Session() as sess:
    print (sess.run(res))
```

填充（Padding）只是以一种奇特的方式来告知进行追加并在输入前添加一些值。在大多数情况下，此值为 0，所以大多数人将其命名为零填充。TensorFlow 支持'VALID'和'SAME'零填充，对于任意填充，需要使用 tf.pad()。'VALID'填充意味着根本没有填充，这意味着输出将与输入具有相同的大小。让我们在同一个例子中用 padding = 1 来计算卷积（注意，对于我们的内核，这是'SAME'填充）。为此，我们只需在数组的开头/结尾添加 1 个零值，即 input = [0,1,0,2,3,0,1,1,0]。在这里，可以注意到你不需要重新计算所有内容——除了第一个/最后一个元素之外，所有元素保持不变。

$1 = 0 \times 2 + 1 \times 1 + 0 \times 3$

$3 = 1 \times 2 + 1 \times 1 + 0 \times 3$

因此，结果是[1, 8, 11, 7, 9, 4, 3]，与 TensorFlow 计算的相同。

```
res = tf.squeeze(tf.nn.conv1d(data, kernel, 1, 'SAME'))
with tf.Session() as sess:
    print sess.run(res)
```

以上是使用步长（Stride）的卷积。步长允许用户在滑动时跳过元素。在之前的所有示例中，我们滑动了 1 个元素，当然也可以一次滑动 s 个元素。因此，如果我们使用前面的示例 padding = 1 并将 stride 更改为 2，则只需获取前一个结果[1,8,11,7,9,4,3]并每次在第二个元素处留下值，得到结果[1,11,9,3]。你可以通过以下方式在 TensorFlow 中执行此操作。

```
res = tf.squeeze(tf.nn.conv1d(data, kernel, 2, 'SAME'))
with tf.Session() as sess:
```

```
    print sess.run(res)
```

7.3.8 二维卷积

函数 tf.nn.conv2d()可以实现二维卷积。在最基本的示例中，没有填充，且 stride = 1。让我们假设你的输入和内核为：

$$I=\begin{pmatrix}4 & 3 & 1 & 0\\ 2 & 1 & 0 & 1\\ 1 & 2 & 4 & 1\\ 3 & 1 & 0 & 2\end{pmatrix} \qquad K=\begin{pmatrix}1 & 0 & 1\\ 2 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}$$

应用卷积运算后，得到以下输出结果：$\begin{pmatrix}14 & 6\\ 6 & 12\end{pmatrix}$

这个输出是通过以下方式计算的：

$$14 = 4\times 1 + 3\times 0 + 1\times 1 + 2\times 2 + 1\times 1 + 0\times 0 + 1\times 0 + 2\times 0 + 4\times 1$$

$$6 = 3\times 1 + 1\times 0 + 0\times 1 + 1\times 2 + 0\times 1 + 1\times 0 + 2\times 0 + 4\times 0 + 1\times 1$$

$$6 = 2\times 1 + 1\times 0 + 0\times 1 + 1\times 2 + 2\times 1 + 4\times 0 + 3\times 0 + 1\times 0 + 0\times 1$$

$$12 = 1\times 1 + 0\times 0 + 1\times 1 + 2\times 2 + 4\times 1 + 1\times 0 + 1\times 0 + 0\times 0 + 2\times 1$$

TensorFlow 的函数 conv2d()分批计算卷积并使用稍微不同的格式。对于输入，格式为[batch, in_height, in_width, in_channels]。对于内核，格式为[filter_height, filter_width, in_channels, out_channels]。所以我们需要以正确的格式提供数据：

```
import tensorflow as tf
k = tf.constant([
    [1, 0, 1],
    [2, 1, 0],
    [0, 0, 1]
], dtype=tf.float32, name='k')
i = tf.constant([
    [4, 3, 1, 0],
    [2, 1, 0, 1],
    [1, 2, 4, 1],
```

```
    [3, 1, 0, 2]
], dtype=tf.float32, name='i')
kernel = tf.reshape(k, [3, 3, 1, 1], name='kernel')
image  = tf.reshape(i, [1, 4, 4, 1], name='image')
```

通过以下方式计算二维卷积：

```
res = tf.squeeze(tf.nn.conv2d(image, kernel, [1, 1, 1, 1], "VALID"))
#VALID 表示没有填充
with tf.Session() as sess:
  print(sess.run(res))
```

输出结果将等同于我们手工计算的那个矩阵。

用一些常量围绕矩阵。在大多数情况下，这种常量为 0，这就是人们称之为零填充的原因。因此，如果你想在我们的原始输入中使用 1 的填充（检查第一个示例，padding = 0，strides = 1），矩阵如下：

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 3 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 \\ 0 & 1 & 2 & 4 & 1 & 0 \\ 0 & 3 & 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

要计算卷积的值，可执行相同的滑动。注意，在我们的情况下，中间的许多值不需要重新计算，因为它们与前面的示例相同。也不会在此显示所有计算，因为这个算法很简单。结果为：

$$\begin{pmatrix} 5 & 11 & 8 & 2 \\ 7 & 14 & 6 & 2 \\ 3 & 6 & 12 & 9 \\ 5 & 12 & 5 & 6 \end{pmatrix}$$

这里：

$5 = 0 \times 1 + 0 \times 0 + 0 \times 1 + 0 \times 2 + 4 \times 1 + 3 \times 0 + 0 \times 0 + 0 \times 1 + 1 \times 1$

……

$6 = 4 \times 1 + 1 \times 0 + 0 \times 1 + 0 \times 2 + 2 \times 1 + 0 \times 0 + 0 \times 0 + 0 \times 0 + 0 \times 1$

TensorFlow 不支持函数 conv2d()中的任意填充，如果需要一些它不支持的填充，可使用函数 tf.pad()。幸运的是，对于我们的输入，填充 SAME 相当于 padding = 1。因此我们几乎不需要对前面的示例做任何改变。

```
res = tf.squeeze(tf.nn.conv2d(image, kernel, [1, 1, 1, 1], "SAME"))
#'SAME'确保输出与输入具有相同的大小,并使用适当的填充
#在我们的例子中填充值为 1
with tf.Session() as sess:
  print sess.run(res)
```

你可以验证答案是否与手动计算的答案相同。

图像处理中的卷积核参数在使用训练的方法调整以前，往往设置成一些固定的值。例如锐化参数：

```
sharpenMatrix = [0.0, -0.2,  0.0, -0.2,  1.8, -0.2, 0.0, -0.2,  0.0]
#使用初始化器(initializer) 初始化卷积核
init = tf.constant_initializer(sharpenMatrix)
W= tf.get_variable('W', shape=[3, 3], initializer=init)
```

完整的代码如下：

```
sharpenMatrix = [0.0, -0.2,  0.0, -0.2,  1.8, -0.2, 0.0, -0.2,  0.0]
init = tf.constant_initializer(sharpenMatrix)
W= tf.get_variable('W', shape=[3, 3], initializer=init)
#添加一个操作来初始化全局变量
init_op = tf.global_variables_initializer()
#在会话中运行计算图
with tf.Session() as sess:
    #运行变量初始化器操作
    sess.run(init_op)
    #评估变量的值
    print(sess.run(W))
```

7.3.9　扩张卷积

扩张卷积（Dilated Convolution）是针对图像语义分割问题中采样会降低图像分辨率、丢失信息而提出的一种卷积思路。

完全卷积表明神经网络只由卷积层组成，没有任何完全连接的层或通常在网络末端找到的多层感知器（MLP）。具有完全连接层的 CNN 就像完全卷积一样，可以端到端地学习。与具有完全连接层的 CNN 的主要区别在于，完全卷积网在任何地方都是学习过滤器，甚至网络末端的决策层都是过滤器。完全卷积网试图学习表示并根据局部空间输入做出决策。附加的完全连接层使得网络能够使用全局信息来学习某些内容，输入的空间布局消失于其中。当不需要用到输入的空间布局时，可以使用附加的完全连接层。

增加原本紧密贴着的卷积核元素之间的距离，但卷积核需要计算的点不变，也就是多余出来的位置全填 0。卷积核的感知区域变大了，又由于卷积核中的有效计算点不变，所以计算量不变。每层的特征图尺寸都不变，所以图像信息都保存了下来。

D-扩张的 $K\times K$ 卷积在进行通常的卷积计算之前扩大卷积核。扩大卷积核，意味着扩大其尺寸，用零填充空位置。实际上，并没有创建扩展的卷积核。相反，卷积核元素（权重）与输入矩阵中的远（不相邻）元素匹配。距离由扩张系数 D 确定。图 7-11 显示了卷积核元素如何与 D-扩张的 3×3 卷积中的输入元素匹配（当卷积核的中心与输入矩阵的中心对齐时）。注意，对于 $D=1$，将获得标准卷积。

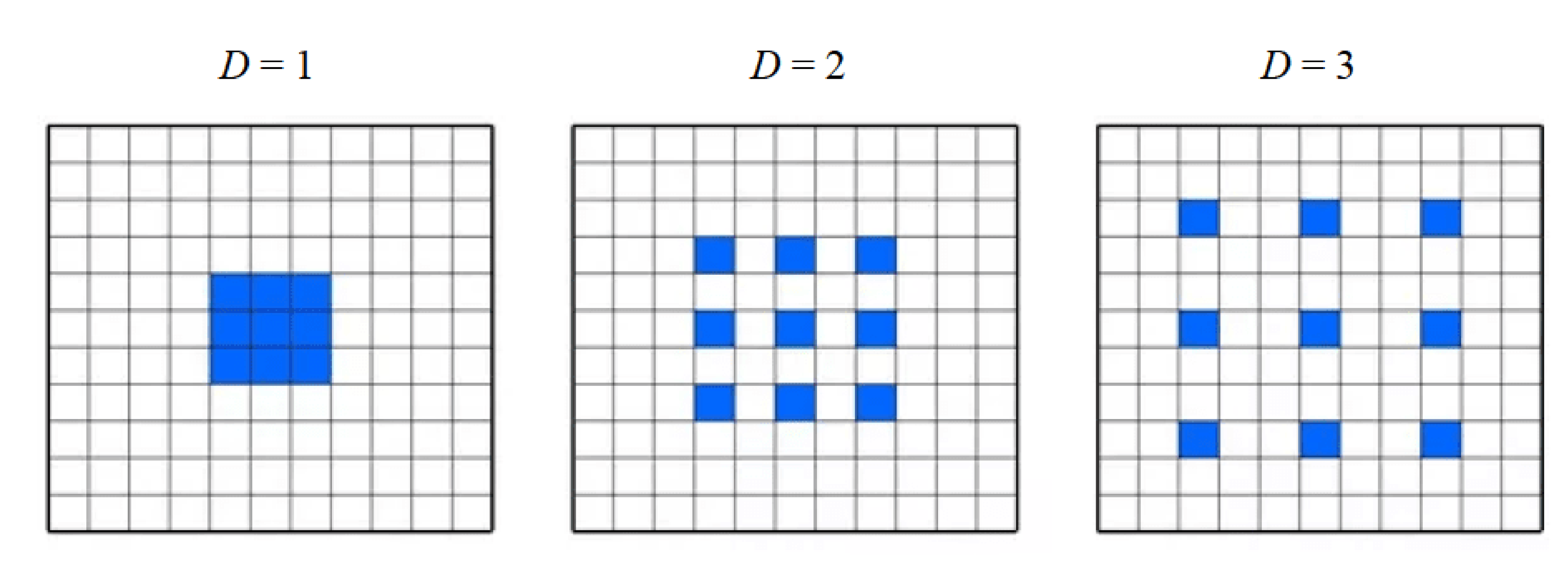


图 7-11　扩张卷积

在 D-扩张卷积中，通常步幅是 1，但也可以使用其他步幅。

7.3.10　TensorFlow 实现简单的语音识别

本小节使用 TensorFlow 识别 10 个英文单词——yes、no、up、down、left、right、on、off、stop、go。

首先使 Git 得到 TensorFlow 源代码，然后在 TensorFlow 源码树下运行训练脚本。

```
#python3 tensorflow/examples/speech_commands/train.py
```

执行上述命令后，会自动开始下载语音指令数据集。这个数据集包含由说话者说 30 个不同单词的超过 105 000 个的 WAVE 文件。这个存档文件容量超过 2GB，因此下载可能需要一段时间，但你应该能够看到进度日志。一旦下载完成，就不用再执行此步骤了。程序会把数据集文件下载到/tmp（临时）目录下。

下载完成后，就将看到下面的训练日志记录信息。

I0730 16:53:44.766740　　55030 train.py:176] Training from step: 1

I0730 16:53:47.289078　　55030 train.py:217] Step #1: rate 0.001000, accuracy 7%, cross entropy 2.611571

这表明初始化过程已经完成，训练循环已经开始。以下是对日志记录信息的详细解释。

- Step #1 表明我们正处于训练循环的第一步。在这个例子中，总共将有 18 000 个步骤，因此您可以查看步骤编号，以了解训练距离完成还有多远。
- 速率 0.001000 是控制网络权重更新速度的学习速率。早期这是一个相对较高的数字，但对于后来的训练周期，它将降低到 0.0001。
- 准确度 7%表示在此训练步骤中有多少类正确预测出来。这个值经常会波动很多，但随着训练的进行，平均值会增加。模型输出一个数字数组，每个数字是输入的预测可能性。通过选择具有最高分数的条目来挑选预测标签，分数始终在 0 和 1 之间，较高的值表示对结果更有信心。

- 交叉熵 2.611571 是我们用来指导训练过程的损失函数的结果。这是通过将当前训练运行的得分向量与正确标签进行比较而获得的得分，并且这个值应该在训练期间呈下降趋势。

经过 100 步后，应该可以看到如下内容。

```
    I0730 16:54:41.813438 55030 train.py:252] Saving to "/tmp/speech commands
train/conv.ckpt-100"
```

这样可以将当前训练的权重保存到检查点文件中。如果训练脚本被中断，可以查找上次保存的检查点，然后使用--start_checkpoint=/tmp/speech_commands_ train/conv.ckpt-100 作为命令行参数从该点开始重新启动脚本。

经过 400 步后，记录的混淆矩阵信息如下：

```
    I0730 16:57:38.073667   55030 train.py:243] Confusion Matrix:
     [[258   0     0     0      0     0    0    0    0    0    0    0]
     [  7    6    26    94      7    49    1   15   40    2    0   11]
     [ 10    1   107    80     13    22    0   13   10    1    0    4]
     [  1    3    16   163      6    48    0    5   10    1    0   17]
     [ 15    1    17   114     55    13    0    9   22    5    0    9]
     [  1    1     6    97      3    87    1   12   46    0    0   10]
     [  8    6    86    84     13    24    1    9    9    1    0    6]
     [  9    3    32   112      9    26    1   36   19    0    0    9]
     [  8    2    12    94      9    52    0    6   72    0    0    2]
     [ 16    1    39    74     29    42    0    6   37    9    0    3]
     [ 15    6    17    71     50    37    0    6   32    2    1    9]
     [ 11    1     6   151      5    42    0    8   16    0    0   20] ]
```

要理解混淆矩阵的含义，首先需要知道正在使用的标签，在这种情况下是"silence", "unknown", "yes", "no", "up", "down", "left", "right", "on", "off", "stop"和 "go"。每列代表一组预测为每个标签的样本，因此第一列代表预测为静音的所有片段，第二列代表所有预测为未知单词的片段，第三列代表 "yes"，依此类推。每行用正确的、真实标签表示音频剪辑，第一行代表所有静音剪辑，第二行代表未知单词，第三行代表"yes"等。

脚本训练完 18 000 步后，会显示一份最终的混淆矩阵和一个根据测试集得出的准

确率得分。如果你按照默认设置进行训练，准确率应该在 80%～90%。

训练完成后，可以执行以下命令行，导出这个语音识别模型。

```
python3 tensorflow/examples/speech commands/freeze.py \
--start checkpoint=/tmp/speech commands train/conv.ckpt-18000 \
--output_file=/home/aaa/speech_commands/my_frozen_graph.pb
```

然后可以用 label_wav.py 脚本，让这个固定的模型识别音频。

```
python3 tensorflow/examples/speech commands/label wav.py \
--graph=/home/aaa/speech commands/my frozen graph.pb \
--labels=/tmp/speech commands train/conv labels.txt \
--wav=/tmp/speech_dataset/left/a5d485dc_nohash_0.wav
```

上述命令应该会输出以下 3 个标签的得分。

```
left (score = 0.79622)
right (score = 0.09350)
_unknown_ (score = 0.07849)
```

希望"left"是最高分，因为这是正确的标签，但由于训练是随机的，可以尝试使用同一文件夹中的一些其他.wav 文件来查看识别效果。标签得分在 0 和 1 之间，较高的值意味着模型对其预测更有信心。

除了返回识别单词的回归值，还可以把识别出来的每个单词发音加注音标。

7.4 nnet3 实现代码

Kaldi 有 3 个独立的深层神经网络代码库，其中开发最活跃的为 nnet3。在 egs/wsj/s5/、egs/rm/s5、egs/swbd/s5 和 egs/hkust/s5b 等示例目录中，可以找到神经网络示例脚本。在运行这些脚本前，必须运行这些目录中“run.sh”的第一阶段以构建用于对齐的系统。

nnet3 设置旨在以自然的方式支持比简单的前馈网络（例如 RNN 和 LSTM 等）更广泛的网络，而不需要任何实际的编码。

7.4.1　数据类型

nnet3 的设置基于一个组件对象，其中一个神经网络是一些组件的堆叠。每个组件对应一层神经网络，我们将单层的褶皱表示为仿射变换，后面跟着非线性组件，所以每层有两个组件。这些旧的组件具有正向传播和反向传播功能，两者都可以在 minibatches 上运行。该网络有一个时间索引的概念，以支持直接作为框架一部分的跨时间特征拼接。这使我们能够通过在网络的中间层包含拼接来支持时延神经网络（TDNN）。

nnet3 通过配置文件定义网络结构。在 nnet3 中，我们有一个通用图结构而不是一个组件的序列。一个 nnet3 神经网络（classNnet）包括以下两个部分。

- 指定组件的列表，没有特定的顺序。
- 一个图结构，其中包含“黏合剂”，指定组件如何组合。

图可以按名称访问组件（这样可以实现某些类型的参数共享）。这“黏合剂”所做的部分工作包括允许时间 t，可以依赖时间 t-1。

以下给出组件和图的一个示例配置文件表示。

```
#组件
component name=affine1 type=NaturalGradientAffineComponent input-dim=48 output-dim=65
component name=relu1 type=RectifiedLinearComponent dim=65
component name=affine2 type=NaturalGradientAffineComponent input-dim=65 output-dim=115
component name=logsoftmax type=LogSoftmaxComponent dim=115
#节点
input-node name=input dim=12
component-node name=affine1_node component=affine1 input=Append(Offset(input, -1), Offset(input, 0), Offset(input, 1), Offset(input, 2))
component-node name=nonlin1 component=relu1 input=affine1 node
component-node name=affine2 component=affine2 input=nonlin1
component-node name=output_nonlin component=logsoftmax input=affine2
output-node name=output input=output_nonlin
```

使用 steps/nnet3/tdnn/make_configs.py 可以自动生成配置文件，生成的脚本看起来

如下：

```
#取自脚本：/egs/tedlium/s5/local/nnet3/run_tdnn.sh
#创建用于 nnet 初始化的配置文件
python steps/nnet3/tdnn/make_configs.py  \
   --feat-dir data/${train_set}_hires \
   --ivector-dir exp/nnet3/ivectors_${train_set} \
   --ali-dir $ali_dir \
   --relu-dim 500 \
   --splice-indexes "-1,0,1 -1,0,1,2 -3,0,3 -3,0,3 -3,0,3 -6,-3,0" \
   --use-presoftmax-prior-scale true \
   $dir/configs || exit 1;
```

图和组件与提供的输入和要求的输出，一起用来构造一个“计算图”（Computation Graph）类。构造计算图是编译过程中的一个重要阶段。计算图是一个非循环图，其中节点对应向量化数值。非循环图的每个节点将由神经网络图的节点（即网络的层）加上一些额外的索引确定。索引包括：时间 t、索引 n，加上一个额外的索引 x。索引 n 表明最小批内的样本编号（例如对于 512 个样本的最小批，编号从 0 到 511）。额外的索引 x 最终可能在卷积方法中用到，但通常为 0。

为将上述内容公式化，把一个索引（Index）定义成一个元组(n,t,x)，还将定义一个 Cindex 元组(node-index,Index)，这里的 node-index（节点索引）是对应到神经网络（即层）中的一个节点的索引。我们实际进行的计算在编译阶段表示为一个 Cindexes 上的有向无环图。

使用神经网络（训练或解码）的过程如下：

- 用户提供计算请求 ComputationRequest 说明可用的是什么，输入的什么索引（例如 time-indexes），要求的是什么输出。
- 计算请求 ComputationRequest 连同一个神经网络一起作为一个 NnetComputation 被编译成一系列的命令。
- NnetComputation 进一步做速度上的优化（可以看成编译上的优化就如同 gcc 的-O 选项）。

- NnetComputer 类负责接收矩阵化输入，评估 NnetComputation 并提供矩阵化输出。可以将 NnetComputer 理解成非常有限的运行时解释语言。

7.4.2 基本数据结构

正如前面提到的，索引（Index）是一个元组(n,t,x)，其中 n 是 minibatch 中的索引，t 是指时间索引，x 是一个供将来使用的占位符索引。在实际的神经网络计算中，1024 将成为一个 1024 维矩阵的列数，索引和矩阵的行之间存在一对一的对应关系。nnet3 的框架不同于 Theano 这样的包，Theano 包使用张量操作，而 nnet3 通过将张量包装成矩阵来有效操作。

如果训练非常简单的前馈网络，索引可能只在 n 维度变化，我们可以随意地将 t 值设置为 0，所以索引看起来像这样：

```
[ (0, 0, 0) (1, 0, 0) (2, 0, 0) ... ]
```

如果我们使用相同类型的网络解码一个句子，索引只会在 t 维度不同，所以会有：

```
[ (0, 0, 0)  (0, 1, 0) (0, 2, 0) ... ]
```

对应于矩阵的行。在网络使用时间信息的情况下，早期的层在训练时需要不同的 t 值，所以我们可能会遇到 n 和 t 不同的索引列表。例如：

```
[ (0, -1,0)  (0, 0, 0)  (0, 1, 0) (1, -1, 0) (1, 0, 0) (1, 1, 0) ...]
```

索引结构体 Index 有默认的排序操作，默认由 n 来排序，然后是 t，最后是 x，所以通常我们也按上面排序。当你看到代码打印索引的向量时，经常看到它们是以紧凑的形式打印，其中可省略 x 索引（如果值为 0），而 t 的范围值以紧凑形式表示，因此上述向量能够写成：

```
[ (0, -1:1) (1,-1:1) ... ]
```

Cindex(int32,Index)，其中 int32 对应神经网络中的一个节点的索引。正如上面提到的，一个神经网络由一组命名组件和一种在节点及节点索引上的图组成。Cindex 在编译过程中使用，它们对应于“计算图”的节点，计算图对应于一个特定的神经网络计

算。Cindex 和特定节点的输出之间有一个对应关系，通常会有 Cindex 和编译计算中矩阵的行之间的一一对应关系。前面提到，有一个索引和矩阵的行之间的对应关系，所不同的是，一个 Cindex 除了告知矩阵为哪一行，也告诉我们为哪个矩阵。假设图中有一个节点“affine1”，要求输出维度 1000 和节点列表编号 2，则 Cindex (2,(0,0,0))相当于列维度 1000 的矩阵的某一行，这将被分配为“affine1”组件的输出。

ComputationGraph（计算图）代表一个 Cindexes 上的有向图，其中每个 Cindex 都有一个它依赖的其他 Cindexes 的列表。在一个简单的前馈结构中，图形将有一个简单的拓扑，该拓扑有多个线性结构，我们可能会有一个依赖于 affine1(0,0,0)的 nonlin1(0,0,0)和依赖于 affine1(1,0,0)的 nonlin1(1,0,0)等。在 ComputationGraph 和其他地方，你会看到称为 cindex_ids 的整数。每一个 cindex_id 都是一系列存储在图中的 Cindexes 的索引，它标识一个特定的 Cindex；cindex_ids 是为了效率起见，作为一个整数比 Cindex 容易使用。

ComputationRequest（计算请求）指定一组命名的输入和输出节点，每个都有一个关联的索引列表。对输入节点，这个列表确定哪些索引要提供给计算；对输出节点，它确定了哪些索引要求被计算。此外，ComputationRequest 包含各种标志，如关于哪些输出/输入节点有需要提供或被要求反向微分，以及模型更新是否执行的信息。

举一个例子，一个 ComputationRequest 可能会指定一个输入节点，命名为“输入”，由索引[(0,-1,0),(0,0,0),(0,1,0)]提供；和一个输出节点，命名为“输出”，与索引[(0,0,0)]被要求。在网络需要左边和右边各一帧上下文的时候，这样会更有意义。其实我们通常在训练中，只要求这样的独立输出帧；在训练中我们通常会有多个 minibatch 的例子，所以索引的 n 维度也有所不同。

计算的目标函数及其输出的导数不是核心神经网络框架的一部分，我们把它交给用户。一般神经网络可能有多个输入和输出节点；这可能在多任务学习或处理多个不同类型的输入数据的框架中是有用的（如多视点学习）。

NnetComputation 代表已经从一个 Nnet 和 ComputationRequest 编译出特定的计算。它包含一个命令序列，每一个都可能是一个传播操作，一个矩阵复制或添加操作，各

种其他简单矩阵命令，例如复制矩阵特定的行从一个矩阵到另一个矩阵；反向传播操作、矩阵大小命令等。计算作用的变量是矩阵的列表，以及可能占据一个矩阵的某些行或列的范围的子矩阵。计算还包含各种索引的集合（整数数组等），这有时需要被作为参数参与特定的矩阵运算。

NnetComputer 对象负责实际执行 NnetComputation。NnetComputer 中的代码实际上很简单（主要是一个带 switch 语句的循环），因为大多数的复杂性发生在 NnetComputation 的编译和优化阶段。

前面的部分概述了一个框架是如何结合在一起的。在这部分，我们将更详细地介绍神经网络本身的结构，以及我们如何把组件结合在一起和表示对从时间 $t-1$ 开始的输入的依赖关系。

nnet3 组件（Component）是一个具有正向传播和反向传播功能的对象。它可能包含参数，也可能只实现一个固定的非线性，例如 Sigmoid 组件。组件接口最重要的部分代码如下：

```
class Component {
   public:
   virtual voidPropagate(const ComponentPrecomputedIndexes *indexes,
                            const CuMatrixBase<BaseFloat> &in,
                            CuMatrixBase<BaseFloat> *out) const = 0;
   virtual voidBackprop(const std::string &debug_info,
                        const ComponentPrecomputedIndexes *indexes,
                        constCuMatrixBase<BaseFloat> &in_value,
                        const CuMatrixBase<BaseFloat> &out_value,
                        const CuMatrixBase<BaseFloat> &out_deriv,
                        Component *to_update, //可能为 NULL;可能与 this 相同或不同
                        CuMatrixBase<BaseFloat> *in_deriv) const = 0;
      ...
};
```

这里，请忽略 ComponentPrecomputedIndexes*indexs 参数。某个特定组件有输入维度和输出维度，这个组件通常会“逐行”转换数据，即 Propagate()方法的输入和输出

矩阵有相同数量的行，输入的每一行被处理后生成相应的输出行。就索引而言，这意味着对应输入和输出的每个元素的索引都是相同的。在 Backprop()方法中也存在类似的逻辑。

一个组件有一个虚函数 Properties()，它会返回包含不同二进制标志位的比特掩码值，这些二进制标志位是定义在 ComponentProperties 的枚举变量。

```
class Component {
   ...
   virtual int32Properties() const = 0;
   ...
};
```

这些属性确定组件的各种特征，如是否包含可更新的参数（kUpdatableComponent）、其传播功能是否支持原位操作（kPropagateInPlace）等。其中很多是优化代码所需的，这样就可以知道可应用哪些优化。

你还会注意到一个 kSimpleComponent 枚举值。如果设置该枚举值，那么这个组件就是“简单”的，这意味着它如同上面的定义一样逐行转换数据。复杂组件允许输入和输出用不同数量的行，而且可能需要知道在输入和输出中使用了什么索引。

Propagate()函数和 Backprop()函数的 const ComponentPrecomputedIndexes 参数只是非简单组件使用的。与 nnet2 框架不同，组件不负责实现诸如跨帧拼接等；相反，我们可以使用描述符（Descriptor）来处理跨帧拼接，具体下面将会解释。

我们先前解释到，一个神经网络是命名组件和“网络节点”上的图的集合，但还没有解释什么是“网络节点”。“网络节点”（NetworkNode）实际上是一个结构体，它有 4 个不同的类别。这 4 个类别由 NodeType 枚举定义：

```
enum NodeType { kInput, kDescriptor, kComponent,kDimRange };
```

其中 3 个较重要的类别是 kInput、kDescriptor 和 kComponent，而 kDimRange 是用来支持将节点的输出分散到各个部分。kComponent 节点是网络的骨干，描述符 kDescriptor 是“黏合剂”将前者组合在一起，支持诸如帧拼接和循环。kInput 节点非常简单，只需要提供一个地方来转储提供的输入和声明输入的维度。你也许会很惊讶，没有 kOutput

节点。原因是，输出节点其实就只是描述符。有一个规则，每个 kComponent 类型节点必须通过它的 kDescriptor 类型的“拥有”节点出现在节点列表前面，这条规则使得图形编译变得更加容易。因此，kDescriptor 类型的节点如果不是有 kComponent 节点紧随其后，就是一个输出节点。为方便起见，Nnet 类有方法 IsOutputNode(int32 node_index) 和 IsComponentInputNode(int32 node_index)可以说明这些区别。

接下来，我们将更加深入神经网络节点的细节。

神经网络可以从配置文件创建。在这里，用一个非常简单的例子来说明配置文件是怎样与描述符关联——这个网络有一个隐藏层，并且在第一个节点的时间轴上做拼接。

```
    #首先是组件
    component  name=affine1  type=NaturalGradientAffineComponent  input-dim=48 
output-dim=65
    component name=relu1 type=RectifiedLinearComponent dim=65
    component  name=affine2  type=NaturalGradientAffineComponent  input-dim=65 
output-dim=115
    component name=logsoftmax type=LogSoftmaxComponent dim=115
    #接下来是节点
    input-node name=input dim=12
    component-node name=affine1 node component=affine1 input=Append(Offset(input, 
-1), Offset(input, 0), Offset(input, 1), Offset(input, 2))
    component-node name=nonlin1 component=relu1 input=affine1_node
    component-node name=affine2 component=affine2 input=nonlin1
    component-node name=output_nonlin component=logsoftmax input=affine2
    output-node name=output input=output_nonlin
```

在配置文件中没有提到描述符（如无“descriptor-node”），而是将“输入”字段作为描述符，如“输入= Append(…)”。配置文件中的每个 component-node 被扩展到两个节点：一个为 kComponent 类型节点，另一个为紧挨着它前面的“输入”字段所定义的 kDescriptor 类型节点。

上面的配置文件没有给出一个 dim-range 节点的例子。这个例子将从 65 个维度的组件 affine1 中取得前 50 个维度，dim-range 节点的基本格式如下：

```
    dim-range-node name=dim-range-node1 input-node=affine1 node dim-offset=0 
dim=50
```

配置文件中的描述符。描述符是一种非常有限的表达式，能够访问定义在图中其他节点的数值。在本节中，我们从其配置文件格式的角度来描述 Descriptors。下面我们将解释描述符如何呈现在代码中。

最简单的描述符（基本情况）只有一个节点名，例如“affine1”（只允许 kComponent 或 kInput 类型的节点出现在这里，为了简化实现）。下面我们将列出可能出现在描述符中的一些类型的表达式，但请记住，这种描述将会给你一个描述符的大概描述，比实际情况更一般化；实际上这些可能只出现在一个特定的层次结构。

```
    #谨慎，这是一种过度生成描述符的简化
    <descriptor>  ::=   <node-name>  ;; kInput 或 kComponent 类型的节点名称
    <descriptor>  ::=  Append(<descriptor>, <descriptor> [, <descriptor> ... ] )
    <descriptor>  ::=   Sum(<descriptor>, <descriptor>)
    <descriptor>  ::=   Const(<value>, <dimension>) ;; 如 Const(1.0, 512)
    <descriptor>  ::=   Scale(<scale>, <descriptor>) ;; 如 Scale(-1.0, tdnn2)
    ;; 例如,故障转移或 IfDefined 可能对 RNN 中的时间 t = -1 有用
    <descriptor>  ::=   Failover(<descriptor>, <descriptor>)
    <descriptor>  ::=   IfDefined(<descriptor>)
    <descriptor>  ::=   Offset(<descriptor>, <t-offset> [, <x-offset> ] )
    ;; 偏移量是整数
    ;; Switch(...)旨在用于发条 RNN 或类似方案
    <descriptor>  ::=   Switch(<descriptor>, <descriptor> [, <descriptor> ...])
    <descriptor>  ::=   Round(<descriptor>, <t-modulus>)  ;; <t-modulus> is an 
integer
    ;; ReplaceIndex 用一个固定的整数<value>替换请求索引中的一些<变量名>(t 或 x)
    ;; 例如,在整合 iVectors 时可能会有用; iVector 总是有时间索引 t = 0
    <descriptor>  ::=   ReplaceIndex(<descriptor>, <variable-name>, <value>)
```

读取描述符的代码尝试以尽可能通用的方式标准化这些描述符，以便几乎所有上述语法都可以读取并转换为内部表示。

```
    ;;; <descriptor> == class Descriptor
    <descriptor> ::=  Append(<sum-descriptor>[, <sum-descriptor> ... ] )
```

```
<descriptor> ::=  <sum-descriptor>  ;; 相当于带有一个参数的 Append()
;;; <sum-descriptor> == class SumDescriptor
<sum-descriptor> ::= Sum(<sum-descriptor>, <sum-descriptor>)
<sum-descriptor> ::= Failover(<sum-descriptor>, <sum-descriptor>)
<sum-descriptor> ::= IfDefined(<sum-descriptor>)
<sum-descriptor> ::= Const(<value>, <dimension>)
<sum-descriptor> ::= <fwd-descriptor>
;;; <fwd-descriptor> == class ForwardingDescriptor
;; <t-offset>和<x-offset>是整数
<fwd-descriptor>  ::=  Offset(<fwd-descriptor>, <t-offset> [, <x-offset> ] )
<fwd-descriptor>  ::=  Switch(<fwd-descriptor>, <fwd-descriptor> [, <fwd-
descriptor> ...])
;; <t-modulus>是一个整数
<fwd-descriptor>  ::=   Round(<fwd-descriptor>, <t-modulus>)
;; <variable-name>是 t 或 x; <value>是一个整数
<fwd-descriptor>  ::=  ReplaceIndex(<fwd-descriptor>,  <variable-name>,
<value>)
;; <node-name>是 kInput 或 kComponent 类型的节点的名称
<fwd-descriptor>  ::=   Scale(<scale>, <node-name>)
<fwd-descriptor>  ::=   <node-name>
```

描述符的设计应该足够严格，使产生的表达式相当容易被计算。当把组件连接到一起时，它们只应该执行繁重的操作，而任何更有趣或非线性的操作都应该在组件中执行。

注意：如果有必要在未知长度的各种索引上做累加或求平均（例如一个文件中的所有 *t* 值），我们倾向于在一个组件内操作——是使用一个复杂的组件，而不是使用描述符。

在代码中从下往上描述描述符。基类 ForwardingDescriptor 处理 Descriptor 类型，该类型将只访问某个单一数值，没有任何 Append (…)或 Sum (…)等的表达。在此接口中较重要的方法是 MapToInput()，实现格式如下：

```
class ForwardingDescriptor {
  public:
```

```
    virtual Cindex MapToInput(const Index &output) const = 0;
    ...
}
```

给定一个特定要求的 Index，该函数将返回一个对应于输入值的 Cindex（引用其他节点）。注意，该函数的参数是一个 Index，而不是一个 Cindex，因为这个数值是不会依赖于自身描述符对应结点的节点索引。有几个 ForwardingDescriptor 的派生类，包括 Simple ForwardingDescriptor（基本情况下，仅持有一个节点索引）、OffsetForwardingDescriptor 和 ReplaceIndexForwardingDescriptor 等。

沿层次结构向上一个级别的是类 SumDescriptor，用于支持表达式 Sum (<desc>，<desc>)、Failover (<desc>，<desc>) 和 IfDefined(<desc>)。很清楚，某个给定索引到 SumDescriptormay 的请求有可能会返回不同的 Cindexes，所以用于 ForwardingDescriptor 的接口行不通。我们还需要支持可选的依赖关系，实现代码如下：

```
class SumDescriptor {
    public:
    virtual void GetDependencies(const Index &ind,
                                 std::vector<Cindex> *dependencies) const = 0;
    ...
};
```

方法 GetDependencies()将所有可能参与计算 Index 数值的 Cindexes 附加到 dependencies 参数。接下来，我们会担心当请求的输入可能是不可计算（例如，因为有限的输入数据或边缘效应）的时候，会发生什么。函数 IsComputable()将进行如下处理：

```
class SumDescriptor {
    public:
    ...
    virtual bool IsComputable(const Index &ind,
                              const CindexSet &cindex_set,
                              std::vector<Cindex> *input_terms) const = 0;
    ...
};
```

这里，CindexSet 对象表示一组 Cindexes 的集合，在这种情况下代表“所有我们知道是可计算的 Cindexes 的集合”。如果这个索引的描述符是可计算的，那么这个方法将返回 true。例如，表达式 Sum (X,Y)只会当 X 和 Y 是可计算的，才可计算。如果这个方法返回 true，那么它还会将 input_terms 附加到实际出现在评估表达式中的输入 Cindexes。例如，在 Failover(X,Y)形式的表达式中，如果 X 是可计算的，那么只会附加 X 到 input_terms，不会附加 Y。

类 Descriptor 是层次结构的顶端。它可以被认为是一个 SumDescriptors 的向量，注意，这个向量长度通常为 1。Descriptor 的功能是附加信息（如附加向量），负责 Append(...)的语法。它有与 SumDescriptor 相同接口的方法 GetDependencies()和 IsComputable()，以及允许用户访问其向量中的各个 Sum 描述符的方法 NumParts()和 Part(int32 n)。

下面详细描述神经网络节点。如上所述，节点有 4 种类型。这 4 种类型由枚举器定义如下：

```
enum NodeType { kInput, kDescriptor, kComponent,kDimRange };
```

实际的 NetworkNode 是一个结构体。为了避免指针的麻烦，又因为 C++不允许联合体内包含类，所以我们有一个略微简化的结构布局。

```
struct NetworkNode {
   NodeType node_type;
   //描述符仅适用于 kDescriptor 类型的节点
   Descriptor descriptor;
   union {
      //对于 kComponent,这个值是进入 Nnet::components_的索引
      int32 component_index;
      //对于 kDimRange,这个值是输入节点的节点索引
      int32 node_index;
   } u;
   //对于 kInput,这个值是输入特征的维度
   //对于 kDimRange,这个值是输出的维度（即范围的长度）
   int32 dim;
   //对于 kDimRange,这个值是进入输入组件的特征的偏移量维度
```

```
    int32 dim_offset;
};
```

不同类型的节点和它们实际使用的成员总结如下。

- kInput 节点只使用 dim。
- kDescriptor 节点只使用 descriptor。
- kComponent 节点只使用 component_index，这索引了 Nnet 的 components_数组。
- kDimRange 节点只使用 node_index、dim 和 dim_offset。

下面我们将给出更多类 Nnet 本身的细节，用于存储整个神经网络。较简单的解释方法是只列出私有数据成员，实现代码如下：

```
class Nnet {
    public:
    …
    private:
    std::vector<std::string> component_names_;
    std::vector<Component*> components_;
    std::vector<std::string> node_names_;
    std::vector<NetworkNode> nodes_;
};
```

上述代码中，component_names_应该有和 components_相同的大小，node_names_应该有 nodes_相同的大小。Nnet 类的实例将名称与组件和节点关联起来。注意，我们自动分配名称到 kDescriptor 类型的节点，通过添加_input 到对应组件节点的名称，这些节点会先于相应的 kComponent 类型的节点。这些 kDescriptor 节点的名称不出现在神经网络的配置文件表示中。

另一个重要的数据类型是结构体 NnetComputation。这代表了一个编译好的神经网络计算，包含一系列的命令和其他必要的解释信息。在内部它定义了许多类型，包括如下的枚举值。

```
enum CommandType {
    kAllocMatrixUndefined, kAllocMatrixZeroed, kDeallocMatrix, kPropagate,
kStoreStats, kBackprop, kMatrixCopy, kMatrixAdd, kCopyRows, kAddRows,
```

```
kCopyRowsMulti, kCopyToRowsMulti, kAddRowsMulti, kAddToRowsMulti, kAddRowRanges,
kNoOperation, kNoOperationMarker };
```

这里特别指出 kPropagate、kBackprop 和 kMatrixCopy 作为命令的自解释例子。有一个结构体 Command 代表一个命令及其参数，大部分的参数都被索引到矩阵和组件的列表。

```
struct Command {
   CommandType command_type;
   int32 arg1;
   int32 arg2;
   int32 arg3;
   int32 arg4;
   int32 arg5;
   int32 arg6;
};
```

还有几个结构体类型定义，用于存储矩阵和子矩阵的大小信息。子矩阵是一个矩阵限制了范围的行和列，可能像 MATLAB 语法：some_matrix(1:10,1:20)。

```
struct MatrixInfo {
   int32 num_rows;
   int32 num_cols;
};
struct SubMatrixInfo {
   int32 matrix_index;  //进入“矩阵”的索引：底层矩阵
   int32 row_offset;
   int32 num_rows;
   int32 col_offset;
   int32 num_cols;
};
```

结构体 NnetComputation 的数据成员如下：

```
struct Command {
   ...
   std::vector<Command> commands;
   std::vector<MatrixInfo> matrices;
```

```
    std::vector<SubMatrixInfo> submatrices;
    //用于 kAddRows、kAddToRows、kCopyRows、kCopyToRows,包含行索引
    std::vector<std::vector<int32> > indexes;
    //用于 kAddRowsMulti、kAddToRowsMulti、kCopyRowsMulti、kCopyToRowsMulti
    //包含(子矩阵索引,行索引)对,或(-1,-1),这意味着对该行不做任何事情
    std::vector<std::vector<std::pair<int32,int32> > > indexes_multi;
    //在 kAddRowRanges 命令中使用的索引,包含(start-index,end-index)对
    std::vector<std::vector<std::pair<int32,int32> > > indexes_ranges;
    //有关神经网络的输入和输出值及衍生物在哪里生活的信息
    unordered_map<int32, std::pair<int32, int32> > input_output_info;
    bool need_model_derivative;
    //以下仅用于非简单组件
    std::vector<ComponentPrecomputedIndexes*> component_precomputed_indexes;
    ...
};
```

称字中带“索引”的向量是如 CopyRows、AddRows 等的矩阵函数参数，它们需要索引向量作为输入（在执行之前，将复制这些向量到 GPU）。

7.5 编译 Kaldi

由 Makefile 定义的目标如下。

- "make depend"将重建依赖关系。在构建工具包之前，运行它是个好主意。如果.depend 文件过期（因为还没有运行"make depend"），则可能会看到如下所示的错误。

```
make[1]: *** No rule to make target '/usr/include/foo/bar', needed by 'baz.o'.  Stop.
```

- "make all"（或"make"）将编译所有代码，包括测试代码。
- "make test"将运行测试代码（用于确保构建能够在用户系统上运行，并且用户没有引入错误）。
- "make clean"将删除所有编译的二进制文件，.o（对象）文件和.a（存档）文件。

- "make valgrind"将运行 valgrind 下的测试程序来检查内存泄漏。
- "make cudavalgrind"将运行测试程序（在 cudamatrix 中），以检查带有 GPU 卡和安装了 CUDA 驱动的操作系统内存泄漏。

编译后的二进制文件在哪里？Makefile 默认不会将编译后的二进制文件放在一个特殊的地方，只是将它们留在相应代码所在的目录中。二进制文件存在于 bin/、gmmbin/、featbin/、fstbin/和 lm/目录中，它们都是 src/的子目录。将来可能会指定一个地方放置所有的二进制文件。

Makefile 如何工作。src/Makefile 文件只是调用所有源子目录（src/base、src/matrix 等）中的 Makefile。这些目录有自己的 Makefile，所有这些都有共同的结构。它们都包括以下行：

```
include ../kaldi.mk
```

这就像 C 语言中的#include 行。在阅读 kaldi.mk 时，请记住它将从实际所在位置下面的一个目录中调用（它位于 src/目录中）。kaldi.mk 文件的示例如下：

```
ATLASLIBS = /usr/local/lib/liblapack.a /usr/local/lib/libcblas.a \
        /usr/local/lib/libatlas.a /usr/local/lib/libf77blas.a
CXXFLAGS = -msse -Wall -I.. \
     -DKALDI_DOUBLEPRECISION=0 -msse2 -DHAVE_POSIX_MEMALIGN \
    -DHAVE_EXECINFO_H=1 -rdynamic -DHAVE_CXXABI_H \
     -DHAVE_ATLAS -I ../../tools/ATLAS/include \
      -I ../../tools/openfst/include \
     -g -O0 -DKALDI_PARANOID
LDFLAGS = -rdynamic
LDLIBS = ../../tools/openfst/lib/libfst.a -ldl $(ATLASLIBS) -lm
CC = g++
CXX = g++
AR = ar
AS = as
RANLIB = ranlib
```

上述这个文件是针对 Linux 操作系统的，我们删除了一些不重要的与 valgrind 相关

的规则。

因此，kaldi.mk 负责设置包含路径、定义预处理器变量、设置编译选项和链接库等。

7.6　端到端深度学习

卷积神经网络在声学模型上的应用源于近些年来其在图像处理上压倒性的成功。最初应用于声学模型的卷积神经网络只有一层或两层，用于特征的提取，再加上 LSTM 和标准的前向网络，代表的例子是 CNN-LSTM-DNN（CLDNN）。由于极其深的卷积神经网络（诸如 VGGNet 和 ResNet）在图像识别上的成功，这些模型也被引入到声学模型，IBM 和微软的学者还在此之上提出了一系列其他变形。这些卷积神经网络模型在 SwitchBoard 这个标准语音识别任务上不断刷新最低错误率的纪录。

7.7　Dropout 解决过度拟合问题

过度拟合是指模型过分地拟合训练样本，但对测试样本预测准确率不高。过度拟合导致模型泛化能力差。Dropout 是一种通过暂时不使用一些节点来解决神经网络过度拟合问题的方法。

将 Dropout 视为一种集成学习形式是有帮助的。在集成学习中，我们采用了一些“较弱”的分类器，分别训练它们，然后在测试时通过平均所有集合成员的响应来使用它们。由于每个分类器都经过单独训练，因此它学会了数据的不同“方面”，并且它们的错误也不同。将它们组合起来有助于产生更强的分级器，不易过度拟合。随机森林（Random Forest）或 GBT（Gradient-Boosted Tree）是典型的集成分类器。

集成学习的一个变体是装袋法，其中集成分类器中的每个成员用输入数据的不同子样本训练，因此仅学习了整个可能的输入特征空间的子集。Dropout 可以被视为装袋法的极端版本。在小批量的每个训练步骤中，Dropout 程序创建不同的网络（通过随机

移除一些单元），其像往常一样使用反向传播进行训练。从概念上讲，整个过程类似于使用许多不同网络的集合（每步一个），每个网络用单个样本训练（即极端装袋）。

在测试阶段，使用整个网络，但权重按比例缩小。在数学上，这近似于整体平均（使用几何平均值作为平均值）。

使用 tf.layers.dropout 实现 Dropout 的代码如下：

```
#导入 MNIST 数据
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=False)
import tensorflow as tf
#训练参数
learning_rate = 0.001
num_steps = 2000
batch_size = 128

#网络参数
num_input = 784      #MNIST 数据输入(img shape: 28*28)
num_classes = 10     #MNIST 类别总数 (0～9 数字)
dropout = 0.25       #暂时不使用一个单位的可能性

#创建神经网络
def conv_net(x_dict, n_classes, dropout, reuse, is_training):
    #定义重用变量的范围
    with tf.variable_scope('ConvNet', reuse=reuse):
        #在多个输入的情况下,TF Estimator 输入是一个字典
        x = x_dict['images']

        #MNIST 数据输入是一个有 784 个特征的一维向量(28 像素×28 像素)
        #重塑以匹配图片格式[高度×宽度×通道数]
        #张量输入变为四维: [批大小,高度,宽度,通道数]
        x = tf.reshape(x, shape=[-1, 28, 28, 1])
        #卷积层有 32 个过滤器,卷积核大小为 5
        conv1 = tf.layers.conv2d(x, 32, 5, activation=tf.nn.relu)
```

```
        #最大池化（下采样），步长为 2，核大小为 2
        conv1 = tf.layers.max_pooling2d(conv1, 2, 2)
        #卷积层有 64 个过滤器，卷积核大小为 3
        conv2 = tf.layers.conv2d(conv1, 64, 3, activation=tf.nn.relu)
        #最大池化（下采样），步长为 2，核大小为 2
        conv2 = tf.layers.max_pooling2d(conv2, 2, 2)
        #为了完全连接层，将数据展平为一维向量
        fc1 = tf.contrib.layers.flatten(conv2)
        #全连接图层（在 tf 的 contrib 文件夹中）
        fc1 = tf.layers.dense(fc1, 1024)
        #应用 Dropout（如果 is_training 为 False，则不应用 dropout）
        fc1 = tf.layers.dropout(fc1, rate=dropout, training=is_training)
        #输出层、类预测
        out = tf.layers.dense(fc1, n_classes)
    return out

#定义模型函数（遵循 TF Estimator 模板）
def model_fn(features, labels, mode):
    #构建神经网络
    #因为 Dropout 在训练和预测时有不同的行为，
    #所以我们需要创建两个仍然共享相同权重的不同计算图
    logits_train = conv_net(features, num_classes, dropout, reuse=False,
                            is_training=True)
    logits_test = conv_net(features, num_classes, dropout, reuse=True,
                           is_training=False)

    #预测
    pred_classes = tf.argmax(logits_test, axis=1)
    pred_probas = tf.nn.softmax(logits_test)
    #如果是预测模式，则提前返回
    if mode == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(mode, predictions=pred_classes)
    #定义损失和优化器
    loss_op = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
        logits=logits_train, labels=tf.cast(labels, dtype=tf.int32)))
```

```
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    train_op = optimizer.minimize(loss_op,
                                  global_step=tf.train.get_global_step())
    #评估模型的准确性
    acc_op = tf.metrics.accuracy(labels=labels, predictions=pred_classes)
    #Estimators 需要返回 EstimatorSpec,然后用它指定训练、评估等的不同操作
    estim_specs = tf.estimator.EstimatorSpec(
        mode=mode,
        predictions=pred_classes,
        loss=loss_op,
        train_op=train_op,
        eval_metric_ops={'accuracy': acc_op})
    return estim_specs
#构建 Estimator
model = tf.estimator.Estimator(model_fn)
#定义用于训练的输入函数
input_fn = tf.estimator.inputs.numpy_input_fn(
    x={'images': mnist.train.images}, y=mnist.train.labels,
    batch_size=batch_size, num_epochs=None, shuffle=True)
#Train the Model
model.train(input_fn, steps=num_steps)
#评估模型
#定义用于评估的输入函数
input_fn = tf.estimator.inputs.numpy_input_fn(
    x={'images': mnist.test.images}, y=mnist.test.labels,
    batch_size=batch_size, shuffle=False)
#使用 Estimator 的 evaluate 方法
e = model.evaluate(input_fn)
print("Testing Accuracy:", e['accuracy'])
```

7.8 矩阵运算

LAPACK（Linear Algebra PACKage）是 Fortran 语言开发的一个线性代数库。BLAS（Basic Linear Algebra Subprograms）是一个 Fortran 90 库，其中包含用于向量-向量运

算、矩阵-向量运算和矩阵-矩阵运算的基本线性代数子程序。CLAPACK 是 BLAS 和 LAPACK 翻译成 C 语言的版本。

Kaldi 中的矩阵代码主要是线性代数库 BLAS 和 LAPACK 上的一个包装。把代码设计成为尽可能灵活地使用它可使用的库。它支持以下 4 种选择。

- ATLAS 是 BLAS 的一个实现加上 LAPACK 的一个子集。
- BLAS 加上 CLAPACK 的一些实现。
- 英特尔的 MKL 提供的 BLAS 和 LAPACK。
- OpenBLAS（http://www.openblas.net）提供 BLAS 和 LAPACK。

程序代码必须知道正在使用这 4 个选项中的哪一个，因为虽然原则上 BLAS 和 LAPACK 是标准化的，但是在接口中有一些差异。Kaldi 代码只需要定义 4 个字符串 HAVE_ATLAS、HAVE_CLAPACK、HAVE_ OPENBLAS 或 HAVE_MKL 中的一个（例如，使用-DHAVE_ATLAS 作为编译器的选项），然后它必须与适当的库链接。

最直接涉及包含外部库和设置适当的 typedef 并定义的代码位于 kaldi-blas.h 中。但是，矩阵代码的其余部分并不完全与这些问题无关，因为 ATLAS 和 CLAPACK 版本的更高级别例程的调用方式不同，所以我们有很多"#ifdef HAVE_ATLAS"指令等。另外，一些例程在 ATLAS 中甚至不可用，所以我们必须自己实现它们。

src 目录中的 configure 脚本负责设置 Kaldi 以使用这些库。它通过在 src 目录中创建文件 kaldi.mk 来完成此操作，kaldi.mk 为编译器提供了相应的标志。例如：

```
    ATLASINC = /home/soft/kaldi/tools/ATLAS/include
    ATLASLIBS = /usr/lib64/atlas/libsatlas.so.3 /usr/lib64/atlas/libtatlas.so.
3 -Wl,-rpath=/usr/lib64/atlas
```

如果不带任何参数调用 configure 脚本，将使用可在系统中“正常”位置找到的任何 ATLAS 安装，但它是可配置的。例如，通过 atlas-root 参数指定 ATLAS 的路径：

```
./configure --atlas-root=../tools/ATLAS/build
```

在 Linux 操作系统下编译 ATLAS：

```
make srcdir=<选择的目录>
```

如果没有给出 srcdir，Makefile 将在 math-atlas 目录中创建一个 TEST 目录（即 srcdir 的值默认为./TEST）。然后切换目录到 srcdir，并执行 make 命令来构建 ATLAS 树。在 ATLAS 树构建后，./atltar.sh 将从 make 命令生成的 ATLAS 源树中构建标准.tar 文件。

第 8 章 语言模型

在语音识别中，使用语言模型解码识别结果。本章先讲解概率语言模型，然后讲解语言模型工具包——KenLM。

8.1 概率语言模型

识别发音“我银行借的已经还了”。假设对于输入语音序列 C“已经还了”，有以下两种识别可能。

S_1： 已经/ 还/ 了/

S_2： 已经/ 黄/ 了/

上述这两种识别结果分别叫作 S_1 和 S_2。如何评价这两种识别结果呢？简化声学模型的计算结果，哪个识别结果更有可能在语料库中出现就选择哪个识别结果。

计算条件概率 $P(S_1|C)$和 $P(S_2|C)$，然后根据 $P(S_1|C)$和 $P(S_2|C)$的值来决定是选择 S_1 还是选择 S_2。

因联合概率 $P(C,S) = P(S|C)P(C) = P(C|S)P(S)$，所以有：

$$P(S \mid C) = \frac{P(C \mid S)P(S)}{P(C)}$$

这也叫作贝叶斯公式。$P(C)$是字串在语料库中出现的概率。比如说语料库中有 1 万个

句子，其中有一句是“生命的意义是什么”，那么 $P(C)=P(\text{"生命的意义是什么"})=0.0001$。在贝叶斯公式中 $P(C)$只是一个用来归一化的固定值，所以实际分词时并不需要计算。

从词串恢复到汉字串的概率只有唯一的一种方式，所以 $P(C|S)=1$。因此，比较 $P(S_1|C)$ 和 $P(S_2|C)$的大小变成比较 $P(S_1)$和 $P(S_2)$ 的大小。也就是说，

$$\frac{P(S_1|C)}{P(S_2|C)}=\frac{P(S_1)}{P(S_2)}$$

因为 $P(S_1)=P(\text{已经,还,了})>P(S_2)=P(\text{已经，黄，了})$，所以选择识别方案 S_1 而不是 S_2。

概率语言模型的任务是评估给定词序列 S 的概率 $P(S)$。那么，如何来表示 $P(S)$呢？为了简化计算，假设每个词之间的概率是与上下文无关的，则：

$$P(S)=P(\omega_1,\omega_2,\cdots,\omega_r)\approx P(\omega_1)\,P(\omega_2)\cdots P(\omega_r)$$

其中，$P(w)$ 就是词 w 出现在语料库中的概率。例如：

$$P(S)=P(\text{已经,还,了})\approx P(\text{已经})\,P(\text{还})\,P(\text{了})$$

词表中的词往往很多，分摊到一个词的概率可能很小，所以 $P(S)$一般是通过很多小数值的连乘积算出来的。如果一个数太小，可能会向下溢出，变成零。例如 0.0000000000000000000000000000001，double 类型表示不出如此小的数。因为函数 $y=\log(x)$，当 x 增大，y 也会增大，所以是单调递增函数。取对数后，表示一个小于 1 的正数的精确度加大了。

$$P(S)\approx P(\omega_1)\,P(\omega_2)\cdots P(\omega_r)\propto \log P(\omega_1)+\log P(\omega_2)+\cdots+\log P(\omega_r)$$

这里的∝是正比符号。因为词的概率小于 1，所以取对数后是负数。最后算 $\log P(w)$，计算任意一个词出现的概率如下：

$$P(\omega_i)=\frac{\omega_i\text{在语料库中的出现次数}n}{\text{语料库中的总词数}N}$$

因此 $\log P(w_i)=\log(freq_w)-\log N$

如果词概率的对数值事前已经算出来了，则结果直接用加法就可以得到 $\log P(S)$，而加法比乘法速度更快。

这个计算 $P(S)$的公式也叫作基于一元概率语言模型的计算公式。

8.1.1 一元模型

假设语料库有 10 000 个词，其中“了”这个词出现了 180 次，则它的出现概率为 0.018，形式化的写法：$P(了)=0.018$。词语概率表如表 8-1 所示。

表 8-1 词语概率表

词　语	词　频	概　率
了	180	0.0180
还	5	0.0005
已经	10	0.0010
黄	2	0.0002

$P(S_1) = P(已经)\ P(还)\ P(了) = 0.001 \times 0.0005 \times 0.018 = 9 \times 10^{-9}$

$P(S_1) = P(已经)\ P(黄)\ P(了) = 0.001 \times 0.0002 \times 0.018 = 3.6 \times 10^{-9}$

可得 $P(S_1) > P(S_2)$，所以选择 S_1 对应的切分。

为了避免向下溢出，取对数的计算结果：

$\log P(S_1) = \log P(已经) + \log P(还) + \log P(了) = -18.526041259610192$

$\log P(S_2) = \log P(已经) + \log P(黄) + \log P(了) = -19.442331991484348$

仍然为：$\log P(S_1) > \log P(S_2)$

8.1.2 数据基础

概率语言模型需要知道哪些是高频词，哪些是低频词。也就是：

$$P(w) = \frac{freq(w)}{全部词的总次数}$$

词语概率表是从语料库统计出来的。为了支持统计中文分词方法，又有分词语料

库。分词语料库内容样例如下：

出国　中介　不能　做　出境游

从分词语料库加工出人工可以编辑的一元词典。因为一元的英文叫法为 Unigram，所以往往把一元词典类叫作 UnigramDic。UnigramDic.txt 中每行一个词及这个词对应的次数，并不存储全部词出现的总次数 totalFreq。UnigramDic.txt 中的样本如下：

有:180

有意:5

意见:10

见:2

分歧:1

大学生:139

生活:1671

一元词典树（Trie）如图 8-1 所示。

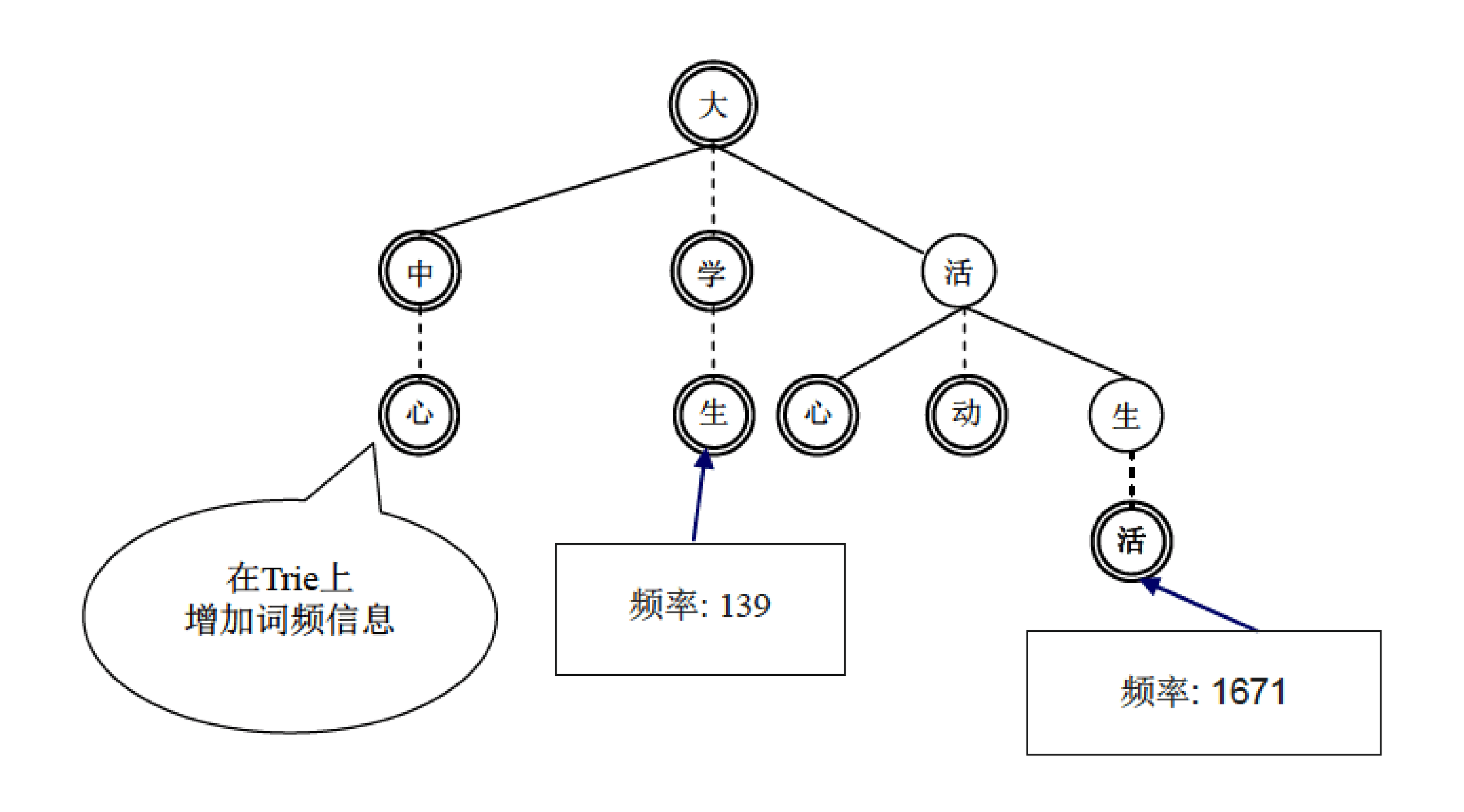


图 8-1　一元词典树

根据 UnigramDic.txt 生成词典树的主要代码如下：

```
while ( ((line = in.readLine()) != null)) {        //逐行读入词典文本文件
   StringTokenizer st = new StringTokenizer(line,"\t");
```

```
        String word = st.nextToken();                    //词
        int freq = Integer.parseInt(st.nextToken());   //次数
        addWord(word,freq);              //把词加入词典树
        totalFreq += freq;               //词的次数加到总次数
    }
```

为了快速生成词典树，可以把词典树的结构保存下来，以便以后直接根据其生成词典树。这里对树中的每个节点编号，并根据编号存储节点之间的引用关系，第一列是节点的编号，第二列是左边孩子节点的编号，第三列是中间孩子节点的编号，第四列是右边孩子节点的编号，最后一列写入节点本身存储的数据。

```
0#1#2#3#有
1#4#5#6#基
2#7#8#9#道
3#10#11#12#羚
4#13#14#15#决
5#16#17#18#诺
```

采用广度优先方式遍历树中的每个节点，同时对每个节点编号。没有孩子节点的分支节点编号设置为-1。

```
TSTNode currentNode = rootNode; //从根节点开始遍历树
int currNodeCode = 0;     //当前节点编号从 0 开始
int leftNodeCode;         //当前节点的左孩子节点编号
int middleNodeCode;       //当前节点的中间孩子节点编号
int rightNodeCode;        //当前节点的右孩子节点编号
int tempNodeCode = currNodeCode;
Deque<TSTNode> queueNode = new ArrayDeque<TSTNode>(); //存放节点数据的队列
queueNode.addFirst(currentNode);
Deque<Integer> queueNodeIndex = new ArrayDeque<Integer>();//存放节点编号的队列
queueNodeIndex.addFirst(currNodeCode);
FileWriter filewrite = new FileWriter(filepath);
BufferedWriter writer = new BufferedWriter(filewrite);
StringBuilder lineInfo = new StringBuilder(); //记录每一个节点的行信息
while (!queueNodeIndex.isEmpty()) {  //广度优先遍历所有树节点,将其加入队列中
    currentNode = queueNode.pollFirst();
                                  //取出队列中第一个节点,同时把它从队列删除
    currNodeCode = queueNodeIndex.pollFirst();
```

```
        //处理左子节点
        if (currentNode.loNode != null) {
            tempNodeCode++;
            leftNodeCode = tempNodeCode;
            queueNode.addLast(currentNode.loNode);
            queueNodeIndex.addLast(leftNodeCode);
        } else {
            leftNodeCode = -1; //没有左孩子节点
        }
        //处理中间子节点
        if (currentNode.eqNode != null) {
            tempNodeCode++;
            middleNodeCode = tempNodeCode;
            queueNode.addLast(currentNode.eqNode);
            queueNodeIndex.addLast(middleNodeCode);
        } else {
            middleNodeCode = -1; //没有中间的孩子节点
        }
        //处理右子节点
        if (currentNode.hiNode != null) {
            tempNodeCode++;
            rightNodeCode = tempNodeCode;
            queueNode.addLast(currentNode.hiNode);
            queueNodeIndex.addLast(rightNodeCode);
        } else {
            rightNodeCode = -1; //没有右边的孩子节点
        }
        lineInfo.delete(0, lineInfo.length());
        lineInfo.append(Integer.toString(currNodeCode) + "#"); //写入当前节点的编号
        lineInfo.append(Integer.toString(leftNodeCode) + "#");
                                                //写入左孩子节点的编号
        lineInfo.append(Integer.toString(middleNodeCode) + "#");
                                                //写入中孩子节点的编号
        lineInfo.append(Integer.toString(rightNodeCode) + "#");
                                                //写入右孩子节点的编号
        lineInfo.append(currentNode.splitChar); // 写入当前节点的分隔字符
        lineInfo.append("\r\n"); //一个节点的信息写入完毕
        writer.write(lineInfo.toString());
```

```
    }
    writer.close();
    filewrite.close();
```

因为当前节点要指向后续节点，所以一开始就预先创建出所有的节点，再逐个填充每个节点中的内容并搭建起当前节点和孩子节点之间的引用关系。读入树结构的代码如下：

```
    TSTNode[] nodeList = new TSTNode[nodeCount]; //创建出节点数组
    //预先创建出所有的节点
    for (int i = 0; i < nodeList.length; ++i) {
       nodeList[i] = new TSTNode();
    }
    while ((lineInfo = reader.readLine()) != null) { //读入一个节点相关的信息
       StringTokenizer st = new StringTokenizer(lineInfo, "#"); //用"#"分隔
       int currNodeIndex = Integer.parseInt(st.nextToken()); //获得当前节点编号
       int leftNodeIndex = Integer.parseInt(st.nextToken()); //获得左子节点编号
       int middleNodeIndex = Integer.parseInt(st.nextToken());
                                                       //获得中子节点编号
       int rightNodeIndex = Integer.parseInt(st.nextToken());
                                                       //获得右子节点编号
       TSTNode currentNode = nodeList[currNodeIndex]; //获得当前节点
       if (leftNodeIndex >= 0) { //从节点数组中取得当前节点的左孩子节点
          currentNode.loNode = nodeList[leftNodeIndex];
       }
       if (middleNodeIndex >= 0) { //从节点数组中取得当前节点的中孩子节点
          currentNode.eqNode = nodeList[middleNodeIndex];
       }
       if (rightNodeIndex >= 0) { //从节点数组中取得当前节点的右孩子节点
          currentNode.hiNode = nodeList[rightNodeIndex];
       }
       char splitChar = st.nextToken().charAt(0); //获取 splitChar 值
       currentNode.splitChar = splitChar; //设置 splitChar 值
    }
```

或者先创建叶节点，再往上创建，直到根节点。实际中，大多使用二进制格式的文件，因为二进制文件比文本文件加载速度更快。生成词典结构的二进制文件 UnigramDic.bin 的

实现代码如下：

```
public static void compileDic(File file) {
   FileOutputStream file_output = new FileOutputStream(file);
   BufferedOutputStream buffer = new BufferedOutputStream(file_output);
   DataOutputStream data_out = new DataOutputStream(buffer);
   TSTNode currNode = root;
   if (currNode == null)
      return;
   int currNodeNo = 1; /* 当前节点编号 */
   int maxNodeNo = currNodeNo;
   /* 用于存放节点数据的队列 */
   Deque<TSTNode> queueNode = new ArrayDeque<TSTNode>();
   queueNode.addFirst(currNode);
   /* 用于存放节点编号的队列 */
   Deque<Integer> queueNodeIndex = new ArrayDeque<Integer>();
   queueNodeIndex.addFirst(currNodeNo);
   data_out.writeInt(nodeCount);     //词典树节点总数
   data_out.writeDouble(totalFreq);  //词频总数
   Charset charset = Charset.forName("utf-8");

   /* 广度优先遍历所有树节点,将其加入数组中 */
   while (!queueNodeIndex.isEmpty()) {
      /* 取出队列第一个节点 */
      currNode = queueNode.pollFirst();
      currNodeNo = queueNodeIndex.pollFirst();

      /* 处理左子节点 */
      int leftNodeNo = 0; /* 当前节点的左孩子节点编号 */
      if (currNode.left != null) {
         maxNodeNo++;
         leftNodeNo = maxNodeNo;
         queueNode.addLast(currNode.left);
         queueNodeIndex.addLast(leftNodeNo);
      }

      /* 处理中间子节点 */
      int middleNodeNo = 0; /* 当前节点的中间孩子节点编号 */
```

```
        if (currNode.mid != null) {
          maxNodeNo++;
          middleNodeNo = maxNodeNo;
          queueNode.addLast(currNode.mid);
          queueNodeIndex.addLast(middleNodeNo);
        }

        /* 处理右子节点 */
        int rightNodeNo = 0; /* 当前节点的右孩子节点编号 */
        if (currNode.right != null) {
          maxNodeNo++;
          rightNodeNo = maxNodeNo;
          queueNode.addLast(currNode.right);
          queueNodeIndex.addLast(rightNodeNo);
        }

        /* 写入本节点的编号信息 */
        data out.writeInt(currNodeNo);
        /* 写入左孩子节点的编号信息 */
        data out.writeInt(leftNodeNo);
        /* 写入中孩子节点的编号信息 */
        data out.writeInt(middleNodeNo);
        /* 写入右孩子节点的编号信息 */
        data out.writeInt(rightNodeNo);
        byte[] splitChar = String.valueOf(currNode.splitChar).getBytes("UTF-8");
        /* 记录 byte 数组的长度 */
        data out.writeInt(splitChar.length);
        /* 写入 splitChar */
        data_out.write(splitChar);

        if (currNode.nodeValue != null) {/* 是结束节点,data 域不为空 */
          CharBuffer cBuffer = CharBuffer.wrap(currNode.nodeValue);
          ByteBuffer bb = charset.encode(cBuffer);
          /* 写入词的长度 */
          data out.writeInt(bb.limit());
          /* 写入词的内容 */
```

```
            for (int i = 0; i < bb.limit(); ++i)
                data_out.write(bb.get(i));
        } else { /* 不是结束节点,data 域为空 */
            data_out.writeInt(0); //写入字符串的长度
        }
    }
    data_out.close();
    file_output.close();
}
```

从二进制文件 UnigramDic.bin 创建词典树，实现代码如下：

```
public static void loadBinaryFile(File file) throws IOException {
    Charset charset = Charset.forName("utf-8"); //得到字符集
    InputStream file_input = new FileInputStream(file);
    /* 读取二进制文件 */
    BufferedInputStream buffer = new BufferedInputStream(file_input);
    DataInputStream data_in = new DataInputStream(buffer);
    /* 获取节点 id */
    nodeCount = data_in.readInt();
    TSTNode[] nodeList = new TSTNode[nodeCount + 1];
    //要预先创建出所有的节点,因为当前节点要指向后续节点
    for (int i = 0; i < nodeList.length; i++) {
        nodeList[i] = new TSTNode();
    }
    /* 读入词典中当前词的个数 */
    totalFreq = data_in.readDouble();
    for (int index = 1;index <= nodeCount;index++) {
        int currNodeIndex = data_in.readInt(); /* 获得当前节点编号 */
        int leftNodeIndex = data_in.readInt(); /* 获得当前节点左子节点编号 */
        int middleNodeIndex = data_in.readInt(); /* 获得当前节点中子节点编号 */
        int rightNodeIndex = data_in.readInt(); /* 获得当前节点右子节点编号 */
        TSTNode currentNode = nodeList[currNodeIndex]; //获得当前节点
        /* 获取 splitChar 值 */
        int length = data_in.readInt();
        byte[] bytebuff = new byte[length];
        data_in.read(bytebuff);
        currentNode.splitChar =
```

```
                    charset.decode(ByteBuffer.wrap(bytebuff)).charAt(0);
        //获取字典中词的内容
        length = data in.readInt();
        /* 如果 data 域不为空,则填充数据域 */
        if (length > 0) {
          bytebuff = new byte[length];
          data in.read(bytebuff);
          String key = new String(bytebuff, "UTF-8"); /* 记录每一个词语 */
          currentNode.nodeValue = key;
        }
        /* 生成树节点之间的对应关系——左、中、右子树 */
        if (leftNodeIndex >= 0) {
          currentNode.left = nodeList[leftNodeIndex];
        }
        if (middleNodeIndex >= 0) {
          currentNode.mid = nodeList[middleNodeIndex];
        }
        if (rightNodeIndex >= 0) {
          currentNode.right = nodeList[rightNodeIndex];
        }
      }
      data in.close();
      buffer.close();
      file input.close();
      root = nodeList[1]; //设置根节点
    }
```

二进制格式的词典文件中是保存词概率取对数后的值，而不是保存词频，所以不会保留词频总数。

```
    public class WordEntry {
      public String word; //词
      public double logProb; //词的概率取对数后的值,也就是 log(P(w))
    }
```

二进制格式的词典文件先写入节点总数，然后写入每个节点的信息。为了方便在 Web 界面上修改词库，可以把词保存到数据库中。创建词表的 SQL 语句如下：

```
    create table AI_BASEWORD  (  //基础词
```

```
    ID            VARCHAR(20)  not null,  //词 ID
    PARTSPEECH    VARCHAR(20),  //词性
    WORD          VARCHAR(200),  //单词
    FREQ          INT,  //词频
    constraint PK WORD BASEWORD primary key (ID)
);
```

从 MySQL 数据库读出词的代码如下：

```
Properties properties = new Properties();
InputStream
is=this.getClass().getResourceAsStream("/database.properties");
properties.load(is);
is.close();
String driver = properties.getProperty("driver");//"com.mysql.jdbc.Driver";
String url = properties.getProperty("url");//"jdbc:mysql://192.168.1.11:3306/
seg?useUnicode=true&amp;characterEncoding=GB2312";
String user = properties.getProperty("user");//"root";
String password = properties.getProperty("password");//"lietu";
Driver drv = (Driver)Class.forName(driver).newInstance();
DriverManager.registerDriver(drv);
Connection con = DriverManager.getConnection(url,user,password);
String sql =("SELECT word, pos, freq FROM AI basewords");
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery(sql);
while (rs.next()){
   String key = rs.getString(1);
   String pos = rs.getString(2);
   int freq = rs.getInt(3);
   addWord(key,pos,freq);  //增加词表到词典树
}
rs.close();
stmt.close();
con.close();
```

8.1.3 改进一元模型

使用更多的信息来改进一元分词。计算从最佳前驱节点到当前节点的转移概率时，

考虑更前面的切分路径。在不改变其他的情况下，用条件概率 $P(w_i|w_{i-1})$的值代替 $P(w_i)$，所以这种方法叫作改进一元分词。如果用最大似然法估计 $P(w_i|w_{i-1})$的值，则有 $P(w_i|w_{i-1}) = freq(w_{i-1},w_i) /freq(w_{i-1})$。假设在二元词表中 $freq$(有,意见)=4，则：

$$P(\text{意见}|\text{有}) \approx freq(\text{有},\text{意见}) /freq(\text{有})=4/4000=0.001$$

可以从语料库中找出 n 元连接，例如，语料库中存在“北京/ 举行/ 新年/ 音乐会/”，则存在一元连接：北京、举行、新年、音乐会，存在二元连接：北京@举行，举行@新年，新年@音乐会。也可以从语料库统计前后两个词一起出现的次数。

由于数据稀疏，导致“意见，分歧”等其他的搭配都没找到。$P(S_1)$和 $P(S_2)$都将是 0，无法通过比较计算结果找到更好的切分方案。这就是零概率问题。

使用 $freq(w_{i-1},w_i)/freq(w_{i-1})$来估计 $P(w_i|w_{i-1})$，使用 $freq(w_{i-2},w_{i-1},w_i)/freq(w_{i-2},w_{i-1})$来估计 $P(w_i|w_{i-2},w_{i-1})$。因为这里采用了最大似然估计，所以把 $freq(w_{i-1},w_i) /freq(w_{i-1})$叫作 $P_{ML}(w_i|w_{i-1})$。

$$P_{l_i}(w_i \mid w_{i-1}) = \lambda_1 P_{ML}(w_i) + \lambda_2 P_{ML}(w_i \mid w_{i-1})$$

$$= \lambda_1 (freq(w_i)/N) + \lambda_2 (freq(w_{i-1},w_i)/freq(w_{i-1}))$$

这里的 $l_1+l_2=1$，且对所有的 i 来说，$l_i \geqslant 0$。N 为语料库的长度。

对于 $P_{l_i}(w_i \mid w_{i-2},w_{i-1})$，则有：

$$P_{l_i}(w_i \mid w_{i-2},w_{i-1}) = \lambda_1 P_{ML}(w_i) + \lambda_2 P_{ML}(w_i \mid w_{i-1}) + \lambda_3 P_{ML}(w_i \mid w_{i-2},w_{i-1})$$

这里 $l_1+l_2+l_3=1$，且对所有的 i 来说，$l_i \geqslant 0$。

根据平滑公式计算，由于：

$$P'(w_i \mid w_{i-1}) = 0.3P(w_i) + 0.7P(w_i \mid w_{i-1})$$

因此，有：

$$\begin{aligned} P(S_1) &= P(\text{有})\, P'(\text{意见}|\text{有})\, P'(\text{分歧}|\text{意见}) \\ &= P(\text{有}) \times (0.3P(\text{意见})+0.7P(\text{意见}|\text{有})) \times (0.3P(\text{分歧})+0.7P(\text{分歧}|\text{意见})) \\ &= 0.0180\times(0.3\times0.001+0.7\times0.001) \times(0.3\times0.0001) \\ &= 5.4\times10^{-9} \end{aligned}$$

$$P(S_2) = P(\text{有意})\, P'(\text{见}|\text{有意})\, P'(\text{分歧}|\text{见})$$
$$= P(\text{有意}) \times (0.3P(\text{见})+0.7P(\text{见}|\text{有意})) \times (0.3P(\text{分歧})+0.7P(\text{分歧}|\text{见}))$$
$$= 0.0005\times(0.3\times0.0002)\times(0.3\times0.0001)$$
$$= 9\times10^{-13}$$

故，$P(S_1) > P(S_2)$。相对于基本的一元模型，改进一元模型的区分度更好。

8.1.4　二元词典

在实际应用中，往往把二元词典类叫作 BigramDic。把“0START.0”和“0END.0”当作两个特殊的词，实现代码如下：

```
public class UnigramDic {
    public final static String startWord="0START.0"; //虚拟的开始词
    public final static String endWord="0END.0"; //虚拟的结束词
}
```

例如：

0START.0@欢迎——“欢迎”是一个开始词。

什么@0END.0——“什么”是一个结束词。

二元词表的格式为“前一个词@后一个词:这两个词组合出现的次数”，例如：

```
中国@北京:100
中国@北海:1
```

二元词表中词条数量很大，至少有几十万条，所以要考虑如何快速查询。快速查找前后两个词在语料库中出现的频次。

可以把二元词表看成是基本词表的常用搭配。两个词的搭配到一个整数值的映射关系，可以用一个 HashMap 表示。

```
public class WordBigram {
    public String left; //左边的词
    public String right; //右边的词
    public WordBigram(String l, String r) {  //构造方法
        left = l;
```

```
        right = r;
    }
    @Override
    public int hashCode() { //散列码
        return left.hashCode() ^ right.hashCode();
    }

    @Override
    public boolean equals(Object o) { //判断两个对象是否相等
        if (o instanceof WordBigram) {
            WordBigram that = (WordBigram) o;
            if (that.left.equals(this.left) && that.right.equals(this.right))
            {
                return true;
            }
        }
        return false;
    }
    public String toString() { //输出内部状态
        return left + "@" + right;
    }
}
```

键为 WordBigram 类型，而值为整数类型。用一个 HashMap 存取两个词的搭配信息：

```
//存放二元连接及对应的频率
HashMap<WordBigram, Integer> bigrams = new HashMap<WordBigram, Integer>();
//存入一个二元连接及对应的频率
bigrams.put(new WordBigram("中国","北京"), 10);
//获取一个二元连接对应的频率
int freq = bigrams.get(new WordBigram("中国","北京"));
System.out.println(freq); //输出 10
```

把相同前缀或相同后缀的词放在一个小的散列表中。把二元词表看成是一个嵌套的映射，用一个嵌套的散列表表示：

```
HashMap<String,HashMap<String,Integer>> bigrams =
```

```
                new HashMap<String,HashMap<String,Integer>>();
    HashMap<String,Integer> val = new HashMap<String,Integer>();
    val.put("北京", 10);
    val.put("上海", 100);
    bigrams.put("中国", val);
    System.out.println(bigrams.get("中国").get("上海"));//输出 100
```

散列表存储一个 String 对象不止 4 字节，而 int 类型为 4 字节。为了节省内存，给每个词编号，用整数代替。这里的 HashMap 往往会有空位置，不是最小完美散列。为了节省内存，用折半查找来查找排好序的数组。

一种实现方法是可以在基本词典树的可结束结点上再挂一个词典树，但这样占用内存多。

另外一种方法是给每个词编号，存储整数到整数的编号，用数组完全展开速度最快。如果有 N 个词，则可以通过如下的方法取得某个二元连接的频率。

```
int N = 20000;
int w1=5; //前一个词的编号
int w2=8; //后一个词的编号
int[][] biFreq = new int[N][N];
int freq = biFreq[w1][w2]; //二元连接的频率
```

分词初始化时，先加载基本词表，对每个词编号，然后加载二元词表，只存储词的编号。

此外，二元词频用开放寻址的散列表也是一个方法。将两个 int 类型混合到一起做键，用 xor。

把搭配信息存放在词典树的叶子节点上，可以看成是一个“键/值”对组成的数组，键为词编号，值为组合频率，实现代码如下：用 BigramMap 表示。

```
public class BigramMap {
    public int[] keys;//词编号
    public int[] vals;//组合频率
}
```

以存储“大学生，生活”为例，“生活”的词编号为 8，“大学生”的词编号为 5。

假设“大学生，生活”的频率为 3，增加二元连接信息后的词典树如图 8-2 所示。

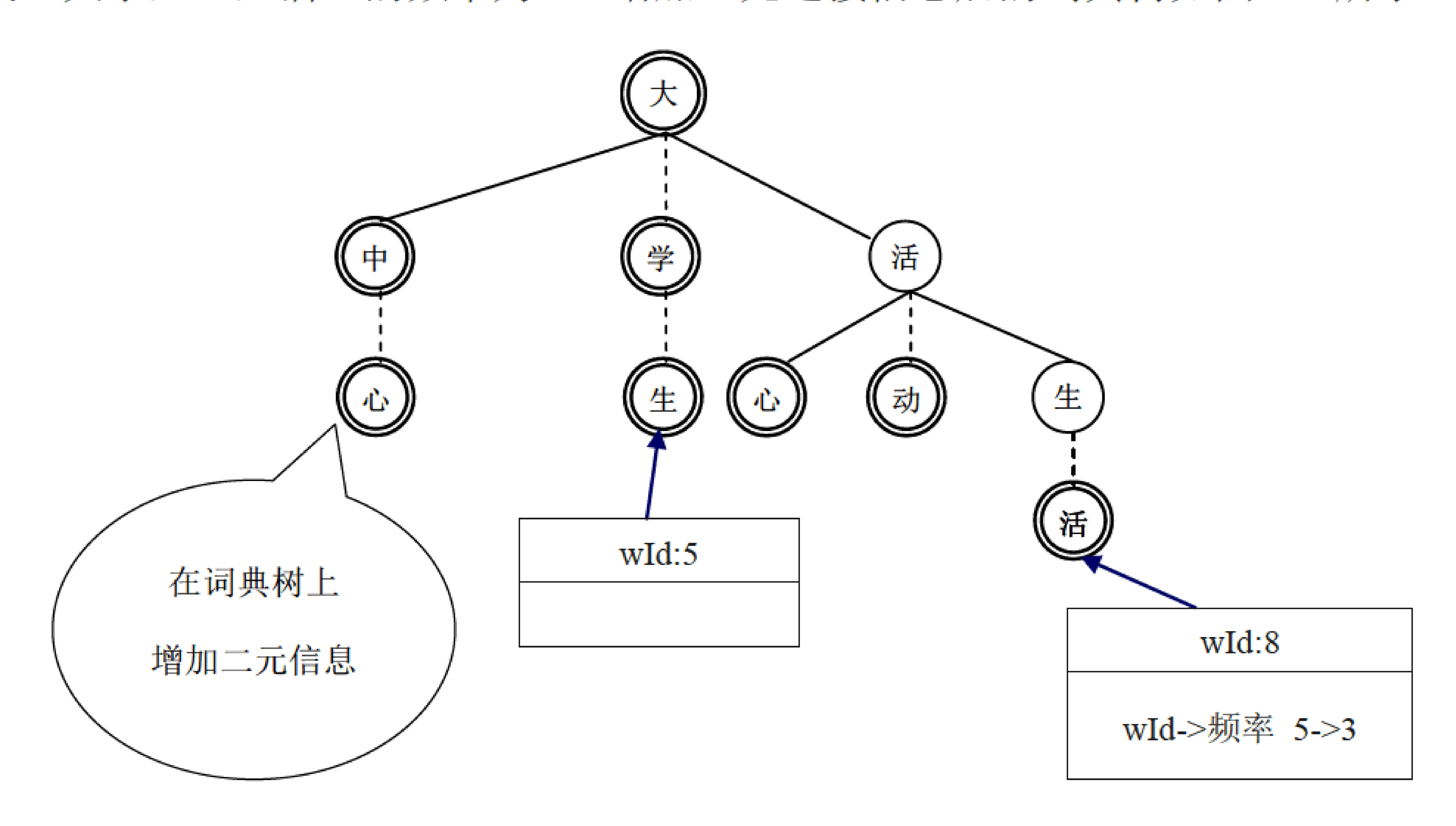


图 8-2 词典树

首先加载基本词典（也就是一元词典），构建词典树结构，然后加载二元词典，也就是在词典树结构上挂二元连接信息后的词典树。

加载基本词典，形成词典树的结构：

```
public TSTNode rootNode;
public double n = 0;          //统计词典中总词频
public int id = 1;            //存储每一个词的 id
public void loadBaseDictionay(String path) throws Exception {
   InputStream file = new FileInputStream(new File(path));
   BufferedReader read = new BufferedReader(new InputStreamReader(file,"GBK"));
   String line = null;
   String pos;
   while ((line = read.readLine()) != null) {
      StringTokenizer st = new StringTokenizer(line, ":");
      String key = st.nextToken();                    //单词文本
      pos = st.nextToken();
      byte code = PartOfSpeech.values.get(pos);   //词性编码
      int frq = Integer.parseInt(st.nextToken()); //单词频率
```

```
        if (rootNode == null) {
            rootNode = new TSTNode(key.charAt(0));
        }
        TSTNode currentNode = getOrCreateNode(key);
        /* 新增节点 */
        if (currentNode.data == null) {
            WordEntry word = new WordEntry(key);
            /* 给新增加词 id */
            word.biEntry.id = id;
            id++; //增加词编号
            /* 统计同一个词的各种词性及对应频率 */
            word.pos.put(code, frq);
            currentNode.data = word;
        } else {
            /* 统计同一个词的各种词性及对应频率 */
            currentNode.data.pos.put(code, frq);
        }
        n += frq; //统计词典中总词频
    }
}
```

加载二元词典。扫描二元连接词典，在词典树中的每个词对应的节点上加上前缀词编号对应的频率。以下是一个整数到整数的键/值对。

```
public void loadBigramDictionay(String path) throws Exception {
    String line = null;
    InputStream file = new FileInputStream(new File(path));
    BufferedReader read = new BufferedReader(new InputStreamReader(file,"GBK"));
    String strline = null;
    String prefixKey = null;       //前缀词
    String suffixKey = null;       //后缀词
    int id = 0;                    //记录单词的 id
    int frq = 0;                   //记录单词的频率
    TSTNode prefixNode = null;     //前缀节点
    TSTNode suffixNode = null;     //后缀节点
    while ((line = read.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(line, ":");
```

```
            strline = st.nextToken();
            //求得@之前的部分
            prefixKey = strline.substring(0, strline.indexOf("@"));
            //求得@之后的部分
            suffixKey = strline.substring(strline.indexOf("@") + 1);
            //寻找后缀节点
            suffixNode = getNode(suffixKey);
            if ((suffixNode == null) || (suffixNode.data == null)) {
              continue;
            }
            //寻找前缀节点
            prefixNode = getNode(prefixKey);
            if ((prefixNode == null) || (prefixNode.data == null)) {
              continue;
            }
            id = prefixNode.data.biEntry.id;            //记录前缀单词的 id
            frq = Integer.parseInt(st.nextToken());     //记录二元频率
            suffixNode.data.biEntry.put(id, frq);
        }
    }
```

建立好词典后，查找二元频率的过程如下：

```
    //从二元词典中查找上下两个词的频率,如果没有,则返回 0
    public int getBigramFreq(WordEntry prev, WordEntry next) {
       //从二元信息入口对象中查找
       if ((next.biEntry != null) && (prev.biEntry != null))
          int frq = next.biEntry.get(prev.biEntry.id);
       if (frq<0)
          return 0;
       return frq;
    }
```

每次都从根节点查找，加载速度慢。把这个词典树保存到一个文件中，以后可以直接从该文件生成树，实现代码如下：

```
    public class BigramDictioanry {
       static final String baseDic = "baseDict.txt";      //基本词典
       static final String bigramDic = "BigramDict.txt";  //二元词典
```

```
    static final String dataDic = "BigramTrie.dat";    //二进制文件
}
```

构造方法：

```
public BigramDictioanry(String dicDir) throws Exception {
    java.io.File dataFile = new File(dicDir + dataDic);

    if (!dataFile.exists()) { //先判断二进制文件是否存在,如果不存在,则创建该文件
        //加载文本格式的基本词典
        loadBaseDictionay(dicDir+ baseDic);

        //加载二元转移关系词典
        loadBigramDictionay(dicDir+ bigramDic);

        //创建二进制数据文件
        createBinaryDataFile(dataFile);
        }
    } else { //从生成的数组树文件加载词典
        loadBinaryDataFile(dataFile);
    }
}
```

8.1.5 完全二叉树数组

词典树的叶结点中存储了词编号和对应的频率。为了节省空间，把键和值都存放在一个数组中。对数组排序后，可以使用折半查找数组，也可以使用完全二叉树实现更快查找，如图 8-3 所示。

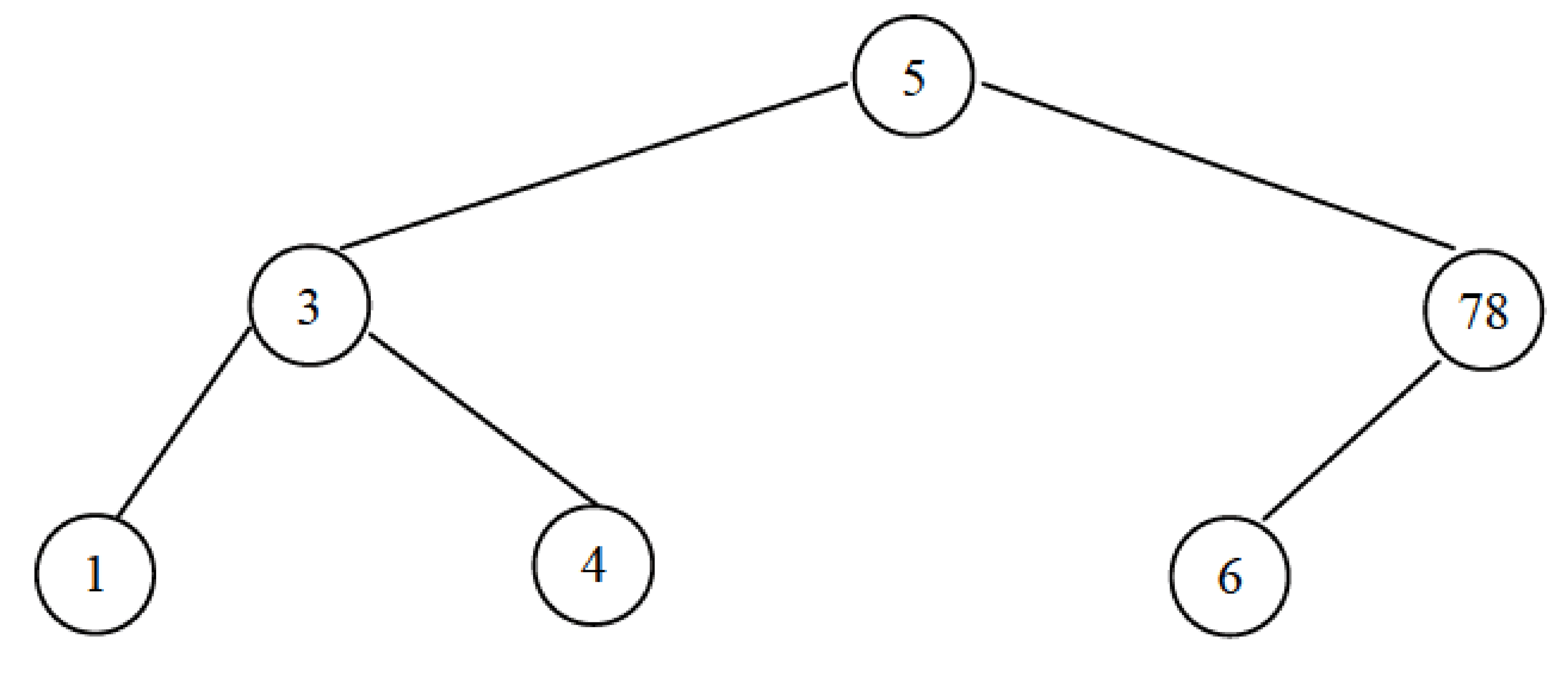


图 8-3 完全二叉树

图 8-3 所示的完全二叉树数组为{5,3,78,1,4,6}。为什么使用完全二叉树？为了不浪费数组中的空间。数组元素不是正好能构成满树，所以只能是完全二叉树。

数组形式存储的完全二叉树：

```
public class CompleteTree {  //完全二叉树
    int[] keys;              //词编号
    int[] vals;              //组合频率
}
```

这里的数组 keys 和数组 vals 中的元素一一对应，也就是说 keys[i]和 vals[i]中的值对应，所以叫作平行数组。

根据给定的数组构建完全二叉树数组。

```
public CompleteTree(int[] k, int[] v) {
    buildArray(k, v);
}
```

根据键查询值的过程比折半查找快。

```
public int find(int data) {                    //查找元素
    int index = 1;                             //从根节点开始查找,根节点编号为 1
    while (index < keys.length) {              //该位置不是空
        if (data < keys[index]) {              //判断要向左查找,还是向右查找
            index = index << 1;                //左子树
        } else if (data == keys[index]) {      //找到了
            return vals[index];                //返回键对应的值
        } else {
            index = (index << 1) + 1;          //右子树
        }
    }
    return -1; //没找到
}
```

完全二叉树比折半查找更快。

```
//对键数组排序，同时值数组也参考键数组调整位置
public static void sortArrays(int[] keys, int[] values) {
    int i, j;
    int temp;
```

```
    //冒泡法排序
    for (i = 0; i < keys.length - 1; i++) {
       //数组最后面已经排好序,所以逐渐减少循环次数
       for (j = 0; j < keys.length - 1 - i; j++) {
          if (keys[j] > keys[j + 1]) {
              temp = keys[j]; //交换键
              keys[j] = keys[j + 1];
              keys[j + 1] = temp;

              temp = values[j]; //交换值
              values[j] = values[j + 1];
              values[j + 1] = temp;
          }
       }
    }
}
```

可以先把所有的元素排好序，元素的编号从 0 开始。对于固定数量的元素，都有一个分配模式。也就是说，如果是一个完全树，则左边应该有多少元素，右边应该有多少元素。

当总共有两个元素时，选择左边的 1 个元素，右边没有，也就是将第 1 个元素作为根节点。当总共有 6 个元素时，选择左边的 3 个元素，右边的 2 个元素，也就是将第 3 个元素作为根节点。

计算完全二叉树的深度，再看底层节点中有几个在根节点的左边。

```
/**
 * 取得完全二叉搜索树编号
 * @param num 节点数
 * @return 根节点编号
 */
static int getRoot(int num) {
   int n = 1;                     //计算满二叉树的节点数
   while (n <= num) {
      n = n << 1;
   }
   int m = n >> 1;
```

```
        int bottom = num - m + 1;      //底层实际节点总数
        int leftMaxBottom = m >> 1;    //假设是满二叉树的情况下,左边节点最大数量
        if (bottom > leftMaxBottom) {  //左边已经填满
            bottom = leftMaxBottom;
        }
        int index = bottom;            //左边的底层节点数
        if(m>1){                       //加上内部的节点数
            index += ((m >> 1) - 1);
        }
        return index;
    }
```

例如，对于下面的数据，测试哪一个作为根节点：

```
int[] data ={1,3,4,5,6};
System.out.println(data[getRoot(data.length)]); //输出 5
```

把 5 作为根节点，这样才能得到一个完全二叉树，如图 8-4 所示。

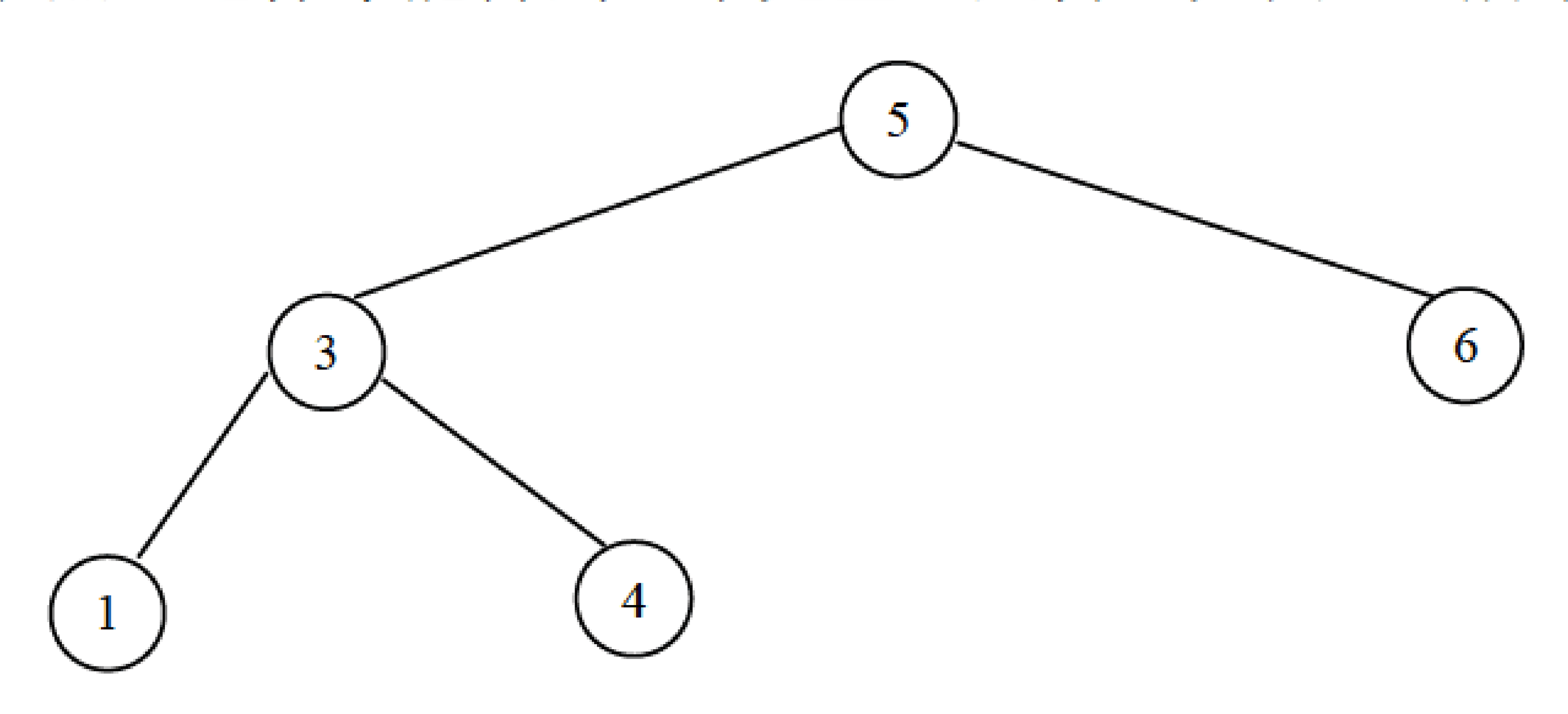


图 8-4　完全二叉搜索树

完全二叉搜索树的任何一个非叶节点的左子树和右子树也都是完全二叉搜索树。所以对于左边的元素和右边的元素可以不断地调用 getRoot 方法。如果要把一个已经生成好的链表形式的二叉树转换成数组形式存放，可以采用广度优先遍历树的方法。要处理的数组范围记录在 Span 类中，实现代码如下：

```
static class Span {
    int start;                     //开始区域
    int end;                       //结束区域
    public Span(int s, int e) {    //构造方法
        start = s;
```

```
        end = e;
    }
}
```

构建完全二叉数组的过程类似广度优先遍历树。首先把根节点放入队列，然后取出队列中的节点，访问这个节点后，把它左边和右边的孩子节点放入队列。采用队列 ArrayDeque 存储要处理的数组范围 Span。构建完全二叉数组的实现代码如下：

```
public void buildArray(int[] keys, int[] values) {
   sortArrays(keys, values);                //先对数组排序
   int pos = 0;                             //已经处理的位置
   this.keys = new int[keys.length];        //完全二叉树数组
   this.vals = new int[keys.length];
   ArrayDeque<Span> queue = new ArrayDeque<Span>();   //堆栈
   queue.add(new Span(0, keys.length));     //加入数组的整个长度
   while (!queue.isEmpty()) {               //如果堆栈中还有元素
      Span current = queue.pop();           //取出元素
      int rootId = CompleteTree.getRoot(current.end - current.start)
            + current.start;
      this.keys[pos] = keys[rootId];
      this.vals[pos] = values[rootId];
      pos++;
      if (rootId > current.start)
        queue.add(new Span(current.start, rootId));
      rootId++;
      if (rootId < current.end)
        queue.add(new Span(rootId, current.end));
   }
}
```

8.1.6 三元词典

对于三元分词，要查找三元词典。三元词典结构可以在二元词典结构上修改——还是键值对，多套一层，还是如同二元词典的 BigramMap 那样，多嵌套一层。

在 BigramMap 中增加一个 IDFreqs[]，实现代码如下：

```
public class BigramMap{
```

```
        public int[] prevIds;          //前缀词 id 集合
        public int[] freqs;            //组合频率集合
        public IDFreqs[] prevGrams;    //前缀元
        public int id;                 //词本身的 id
    }
```

二级前驱词：

```
    public class IDFreqs{
        int[] ids; //词编号
        int[] freqs; //次数
    }
```

布隆过滤器存储 n 元连接。如果内存放不下，可以把 n 元连接存储在 B+树结构的嵌入式数据库中。

8.1.7　*N* 元模型

为了切分更准确，要考虑一个词所处的上下文。例如，“上海银行间的拆借利率上升”，因为“银行”后面出现了“间”这个词，所以把“上海银行”分成“上海”和“银行”两个词。

一元分词假设前后两个词的出现概率是相互独立的，但实际不太可能。例如，沙县小吃附近经常有桂林米粉，所以“沙县小吃”和“桂林米粉”这两个词是正相关。但是很少会有人把“沙县小吃”和“星巴克”相提并论。再如，[羡慕][嫉妒][恨]这 3 个词有时会连续出现，切分出来的词序列越通顺，越有可能是正确的切分方案。N 元模型使用 n 个单词组成的序列来衡量切分方案的合理性。

估计单词 w_1 后出现 w_2 的概率。根据条件概率的定义：

$$P(w_2 \mid w_1) = \frac{P(w_1, w_2)}{P(w_1)}$$

可以得到：$P(w_1,w_2)= P(w_1)P(w_2|w_1)$

同理：$P(w_1,w_2,w_3)= P(w_1,w_2)P(w_3|w_1,w_2)$

所以有：$P(w_1,w_2,w_3)= P(w_1)P(w_2|w_1)P(w_3|w_1,w_2)$

更加一般的形式：

$P(S)=P(w_1,w_2,\cdots,w_n)= P(w_1)P(w_2|w_1)P(w_3|w_1,w_2)\cdots P(w_n|w_1w_2\cdots w_{n-1})$

这叫作概率的链规则。其中，$P(w_2|w_1)$表示 w_1 之后出现 w_2 的概率。如果词 w_1 和 w_2 独立出现，则 $P(w_2|w_1)$等价于 $P(w_2)$。这样需要考虑在 n-1 个单词序列后出现单词 w 的概率。直接使用这个公式计算 $P(S)$存在两个致命的缺陷：一个缺陷是参数空间过大，不可能实用化；另一个缺陷是数据稀疏严重。例如，当词汇量（V）为 20 000 时，可能的二元组合数量有 400 000 000 个，可能的三元组合数量有 8 000 000 000 000 个，可能的四元组合数量有 1.6×10^{17} 个。

为了解决这个问题，我们引入了马尔科夫假设，即一个词的出现仅仅依赖于它前面出现的有限的一个或几个词。如果简化成一个词的出现仅依赖于它前面出现的一个词，那么就称为二元模型（Bigram），即

$$P(S) = P(w_1,w_2,\cdots,w_n)= P(w_1)\, P(w_2|w_1)\, P(w_3|w_1,w_2)\cdots P(w_n|w_1w_2\cdots w_{n-1})$$
$$\approx P(w_1)\, P(w_2|w_1)P(w_3|w_2)\cdots P(w_n|w_{n-1})$$

例如，

$P(S_1) = P(\text{有})\ P(\text{意见}|\text{有})\ P(\text{分歧}|\text{意见})$

如果简化成一个词的出现仅依赖于它前面出现的两个词，就称为三元模型（Trigram）。如果一个词的出现不依赖于它前面出现的词，就称为一元模型（Unigram），也就是已经讲解过的概率语言模型分词方法。

如果切分方案 S 是由 n 个词组成的，那么 $P(w_1)\ P(w_2|w_1)P(w_3|w_2)\cdots P(w_n|w_{n-1})$也是 n 项连乘积。无论是采用一元模型、二元模型还是采用三元模型，都是 n 项连乘积。只不过二元以上模型是条件概率的连乘积。例如，对于切分“有意见分歧”来说，二元模型计算：$P(\text{有})\ P(\text{意见}|\text{有})\ P(\text{分歧}|\text{意见})$，三元模型计算：$P(\text{有})\ P(\text{意见}|\text{有})\ P(\text{分歧}|\text{有},\text{意见})$。

因为 $P(w_i|w_{i-1}) = freq(w_{i-1},w_i)\ /freq(w_{i-1})$，所以二元分词不仅用到二元词典，还需要用到一元词典。

8.1.8 生成语言模型

先有语料库，后有词典文件。如果输入串是“迈向　充满　希望　的　新　世纪”，则返回“迈向@充满”“充满@希望”“希望@的”“的@新”“新@世纪”5 个二元连接。再加上虚拟的开始词和结束词，分别为“0START.0@迈向”和“世纪@0END.0”。

以下代码用于找到切分语料库中所有的二元连接串。

```
FileInputStream file = new FileInputStream(new File(fileName));
BufferedReader buffer = new BufferedReader(new InputStreamReader(file,"GBK"));
BufferedWriter result = new BufferedWriter(new FileWriter(resultFile,true));
String line;
while ((line = buffer.readLine()) != null) { //按行处理
    if (line.equals(""))
        continue;
    StringTokenizer st = new StringTokenizer(line," " ); //用空格分开
    String prev = st.nextToken(); //取得上一个词
    if(!st.hasMoreTokens())   {
        continue;
    }
    String next = st.nextToken(); //取得下一个词
    if(!st.hasMoreTokens())   {
        continue;
    }
    while (true) {
        String bigramStr = prev + "@" + next; //组成一个二元连接
        result.write(bigramStr);   //把二元连接串写入结果文件
        result.write("\r\n");
        if(!st.hasMoreTokens()){   //如果没有更多的词,就退出
          break;
        }
        prev = next;               //下一个词作为上一个词
        next = st.nextToken();     //得到下一个词
    }
}
result.close();                    //关闭写入文件
```

因为词是先进先出的，所以一个 n 元连接用一个容量为 n 的队列表示。当为只有

固定长度的队列添加一个元素时，队列会溢出固定大小。也就是说，这个队列不能保留所有的元素，会自动移除最老的元素。实现一个三元连接的代码如下：

```
CircularQueue q = new CircularQueue(3); //容量为 3 的队列
q.add("迈向");
q.add("充满");
q.add("希望");
q.add("的");
q.add("新");
q.add("世纪");
Iterator it = q.iterator();
//因为 q 中只保留了 3 个词,所以只返回 3 个词
while (it.hasNext()) {
   Object word = it.next();
   System.out.print(word+" ");
}
```

输出结果：

```
的 新 世纪
```

统计 *n* 元概率的项目（https://github.com/esbie/ngrams）。首先从人民日报切分语料库得到新闻行业语言模型，然后切分行业文本得到垂直语料库，最后根据垂直语料库统计出垂直语言模型，这样可以提升语言模型准确度。

8.1.9 评估语言模型

通过困惑度（Perplexity）来衡量语言模型。困惑度是和一个语言事件的不确定性相关的度量。考虑词级别的困惑度，“行”后面可以跟的词有“不行”“代码”“善”“走”，所以“行”的困惑度较高。但有些词不太可能跟在“行”后面，例如“您”“分”。而有些词的困惑度比较低，例如“康佳”等专有名词，后面往往跟着“彩电”等词。语言模型的困惑度越低越好，相当于有比较强的消除歧义能力。如果从更专业的语料库学习出语言模型，则有可能获得更低的困惑度，因为专业领域中的词搭配更加可预测。

1. 困惑度的定义

假设有一些测试数据，n 个句子：$S_1,S_2,S_3,\cdots,S_n$，计算整个测试集 T 的概率：

$$\log\sum_{i=1}^{n}P(s_i)=\sum_{i=1}^{n}\log P(s_i)$$

困惑度 Perplexity(t)=2^{-x}，这里的 $x=\frac{1}{W}\sum_{i=1}^{n}\log P(S_i)$，$W$ 是测试集 T 中的总词数。

2. 困惑度的构想

假设有个词表 V，其中有 N 个词，形式化的写法为 $N=|V|$。模型预测词表中任何词的概率都是 $P(w)=(1/N)$。很容易计算这种情况下的困惑度 Perplexity(t)=2^{-x}，这里 $x=\log\frac{1}{N}$，所以 Perplexity(t)=N。困惑度是对有效分支系数的衡量。

例如，训练集有 3800 万个词，来自华尔街日报（WSJ），词表有 19,979 个词，测试集有 150 万个词，也来自华尔街日报，求得一元模型的困惑度值为 962，二元模型的困惑度值为 170，三元模型的困惑度值为 109。

8.1.10 平滑算法

语料是有限的，不可能覆盖所有的词汇。比如说 N 元模型，当 N 较大的时候，由于样本数量有限，导致很多的先验概率值都是 0，这就是零概率问题。当 n 值为 1 的时候，也存在零概率问题，也就是说一元模型中也存在零概率问题。例如一些词在词表中，却没有出现在语料库中，这说明语料库太小了，没能包括一些本来可能出现的词的句子。

做过物理实验的都知道，我们一般测量几个点后，就可以画出一条大致的曲线，这叫作回归分析。利用这条曲线，可以修正测量的一些误差，还可以估计一些没有测量过的值。平滑算法用观测到的事件来估计未观察到事件的概率。例如从那些比较高的概率值中匀一些给那些低的或是 0 的。为了更合理的分配概率，可以根据整个直方图分布曲线去猜那些为 0 的实际值应该是多少。

由于训练模型的语料库规模有限且类型不同，许多合理的搭配关系在语料库中不一定出现，因此会造成模型出现数据稀疏现象。数据稀疏在统计自然语言处理中的一个表现就是零概率问题。有各种平滑算法来解决零概率的问题。例如，我们对自己能做到的事情比较了解，而不太了解别人是否能做到一些事情，这样导致高估自己而低估别人。所以需要开发一个模型减少已经看到事件的概率，而允许没有看到的事件发生。

平滑有黑盒平滑方法和白盒平滑方法两种。黑盒平滑方法把一个项目作为不可分割的整体。而白盒平滑方法把一个项目作为可分拆的，可用于 n 元模型。

加法平滑算法是最简单的一种平滑算法。加法平滑算法的原理是给每个项目增加 lambda(1>=lambda>=0)，然后除以总数作为项目新的概率。因为数学家拉普拉斯（laplace）首先提出用加 1 的方法估计没有出现过的现象的概率，所以加法平滑也叫作拉普拉斯平滑。

以下是加法平滑算法的一个实现。

```
//根据原始的计数器生成平滑后的分布
public static<E> Distribution<E> laplaceSmoothedDistribution(
                                    GenericCounter<E> counter,
                                    int numberOfKeys,
                                    double lambda) {
    Distribution<E> norm = new Distribution<E>();//生成一个新的分布
    norm.counter = new Counter<E>();
    double total = counter.totalDoubleCount();//原始的出现次数
    double newTotal = total + (lambda * (double) numberOfKeys);//新的出现次数
//有多大可能性出现零概率事件
    double reservedMass =
                ((double) numberOfKeys - counter.size())* lambda / newTotal;
    norm.numberOfKeys = numberOfKeys;
    norm.reservedMass = reservedMass;
    for (E key : counter.keySet()) {
      double count = counter.getCount(key);
      //对于任何一个词来说,新的出现次数是原始出现次数加 lambda
      norm.counter.setCount(key, (count + lambda) / newTotal);
    }
    if (verbose) {
```

```
        System.err.println("unseenKeys=" + (norm.numberOfKeys - norm.counter.
size() + " seenKeys=" + norm.counter.size() + " reservedMass=" + norm.reservedMass);
        System.err.println("0 count prob: " + lambda / newTotal);
        System.err.println("1 count prob: " + (1.0 + lambda) / newTotal);
        System.err.println("2 count prob: " + (2.0 + lambda) / newTotal);
        System.err.println("3 count prob: " + (3.0 + lambda) / newTotal);
      }
      return norm;
    }
```

需要注意的是，reservedMass 是所有零概率词出现概率的总和，而不是其中某个词出现概率的总和。取得指定 key 的概率实现代码如下：

```
   public double probabilityOf(E key) {
      if (counter.containsKey(key)) {
        return counter.getCount(key);
      } else {
        int remainingKeys = numberOfKeys - counter.size();
        if (remainingKeys <= 0) {
          return 0.0;
        } else {
          //如果有零概率的词,
          //则这个词的概率是 reservedMass 分摊到每个零概率词的概率
          return (reservedMass / remainingKeys);
        }
      }
    }
```

上述这种方法中的 lambda 值不好选取，在接下来要讲解的另一种平滑算法 Good-Turing 方法中则不需要 lambda 值。为了讲解 Good-Turing 方法，首先假设词典中共有 x 个词，在语料库中出现 r 次的词有 N_r 个。例如，出现一次的词有 N_1 个，则语料库中的总词数 $N=0\times N_0+1\times N_1+r\times N_r+\cdots$，而 $x=N_0+N_1+N_r+\cdots$。然后，使用观察到的类别 $r+1$ 的全部概率去估计类别 r 的全部概率。计算中的第一步是估计语料库中没有出现过的词的总概率 $p_0 = N_1 / N$，分摊到每个词的概率为 $N_1 / (N \cdot N_0)$。第二步是估计语料库中出现过一次的词的总概率 $p_1 = 2N_2 / N$，分摊到每个词的概率为 $2N_2/ (N \cdot N_1)$。依次类推，


当 r 值比较大时，N_r 可能为 0，这时不再求平滑。词的概率图如图 8-5 所示。

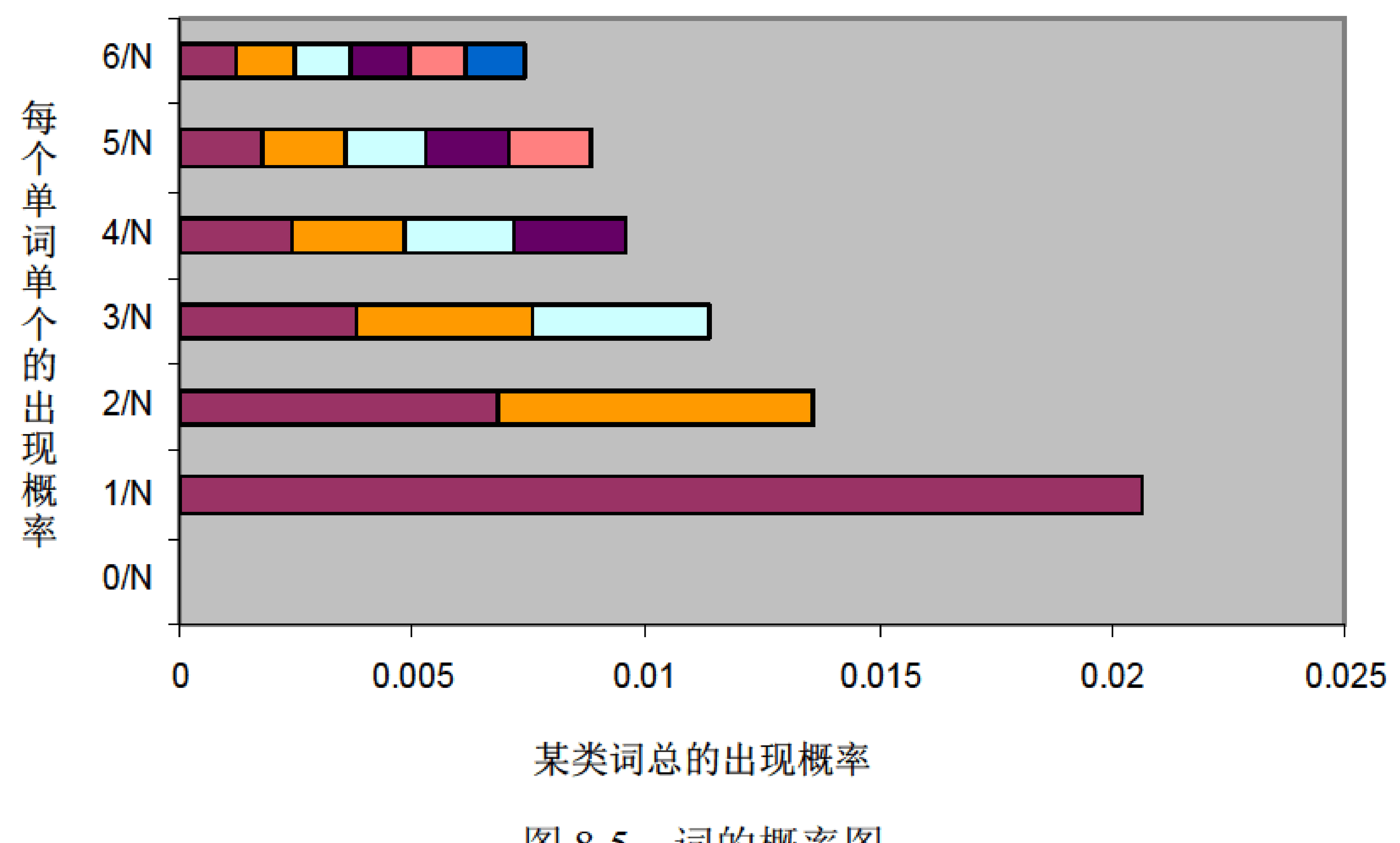


图 8-5　词的概率图

Good-Turing 平滑算法实现代码如下：

```
public static <E> Distribution<E> goodTuringSmoothedCounter(
                                        GenericCounter<E> counter,
                                        int numberOfKeys) {
    //收集计数数组,也就是直方图
    int[] countCounts = getCountCounts(counter);
    //如果计数数组不可靠,就不要用 Good-Turing 方法
    //而采用拉普拉斯平滑方法
    for (int i = 1; i <= 10; i++) {
      if (countCounts[i] < 3) {
        return laplaceSmoothedDistribution(counter, numberOfKeys, 0.5);
      }
    }
    double observedMass = counter.totalDoubleCount();
    double reservedMass = countCounts[1] / observedMass;

    //计算和缓存调整后的频率,同时也调整观察到的项目总数
    double[] adjustedFreq = new double[10];
    for (int freq = 1; freq < 10; freq++) {
```

```
      adjustedFreq[freq] = (double) (freq + 1) * (double) countCounts[freq + 1] /
                               (double) countCounts[freq];
      observedMass -= ((double) freq - adjustedFreq[freq]) * countCounts[freq];
    }
    double normFactor = (1.0 - reservedMass) / observedMass;
    Distribution<E> norm = new Distribution<E>();
    norm.counter = new Counter<E>();
    //填充新的分布,同时重新归一化
    for (E key : counter.keySet()) {
      int origFreq = (int) Math.round(counter.getCount(key));
      if (origFreq < 10) {
        norm.counter.setCount(key, adjustedFreq[origFreq] * normFactor);
      } else {
        norm.counter.setCount(key, (double) origFreq * normFactor);
      }
    }
    norm.numberOfKeys = numberOfKeys;
    norm.reservedMass = reservedMass;
    return norm;
  }
```

对条件概率的 N 元估计平滑：

$$P_{\mathrm{GT}}(w_i \mid w_1,\cdots,w_{i-1}) = \frac{c^*(w_1,\cdots,w_i)}{c^*(w_1,\cdots,w_{i-1})}$$

这里的 c^*来源于 GT 估计，也就是 Good-Turing 估计。

估计三元条件概率：

$$P_{\mathrm{GT}}(w_3 \mid w_1,w_2) = \frac{c^*(w_1,w_2,w_3)}{c^*(w_1,w_2)}$$

对于一个没出现过的，三元联合概率为：

$$P_{\mathrm{GT}}(w_1,w_2,w_3) = \frac{c_0^*}{N} = \frac{N_1}{N_0 \times N}$$

对于一元和二元模型，也是如此。N 元分词中，没有在词表中出现的单字都要根据 GT 估计给一个概率。

8.2　KenLM 语言模型工具包

在 Linux 操作系统下安装：

```
pip install https://github.com/kpu/kenlm/archive/master.zip
```

下载测试语言模型文件：

```
$git clone https://github.com/kpu/kenlm.git
```

测试语言模型文件 test.arpa 的内容如下：

```
#cat ./test.arpa

\data\
ngram 1=37
ngram 2=47
ngram 3=11
ngram 4=6
ngram 5=4

\1-grams:
-1.383514      ,              -0.30103
-1.139057      .              -0.845098
-1.029493      </s>
-99            <s>            -0.4149733
-1.995635      <unk>          -20
-1.285941      a              -0.69897
-1.687872      also           -0.30103
-1.687872      beyond         -0.30103
-1.687872      biarritz       -0.30103
-1.687872      call           -0.30103
-1.687872      concerns       -0.30103
-1.687872      consider       -0.30103
-1.687872      considering    -0.30103
-1.687872      for            -0.30103
-1.509559      higher         -0.30103
-1.687872      however        -0.30103
-1.687872      i              -0.30103
```

```
-1.687872      immediate    -0.30103
-1.687872      in           -0.30103
-1.687872      is           -0.30103
-1.285941      little       -0.69897
-1.383514      loin         -0.30103
-1.687872      look         -0.30103
-1.285941      looking      -0.4771212
-1.206319      more         -0.544068
-1.509559      on           -0.4771212
-1.509559      screening    -0.4771212
-1.687872      small        -0.30103
-1.687872      the          -0.30103
-1.687872      to           -0.30103
-1.687872      watch        -0.30103
-1.687872      watching     -0.30103
-1.687872      what         -0.30103
-1.687872      would        -0.30103
-3.141592      foo
-2.718281      bar          3.0
-6.535897      baz          -0.0

\2-grams:
-0.6925742      , .
-0.7522095      , however
-0.7522095      , is
-0.0602359      . </s>
-0.4846522      <s> looking  -0.4771214
-1.051485       <s> screening
-1.07153        <s> the
-1.07153        <s> watching
-1.07153        <s> what
-0.09132547     a little     -0.69897
-0.2922095      also call
-0.2922095      beyond immediate
-0.2705918      biarritz .
-0.2922095      call for
-0.2922095      concerns in
-0.2922095      consider watch
```

```
-0.2922095      considering consider
-0.2834328      for ,
-0.5511513      higher more
-0.5845945      higher small
-0.2834328      however ,
-0.2922095      i would
-0.2922095      immediate concerns
-0.2922095      in biarritz
-0.2922095      is to
-0.09021038     little more      -0.1998621
-0.7273645      loin ,
-0.6925742      loin .
-0.6708385      loin </s>
-0.2922095      look beyond
-0.4638903      looking higher
-0.4638903      looking on       -0.4771212
-0.5136299      more .           -0.4771212
-0.3561665      more loin
-0.1649931      on a             -0.4771213
-0.1649931      screening a      -0.4771213
-0.2705918      small .
-0.287799       the screening
-0.2922095      to look
-0.2622373      watch </s>
-0.2922095      watching considering
-0.2922095      what i
-0.2922095      would also
-2      also would      -6
-15     <unk> <unk>     -2
-4      <unk> however   -1
-6      foo bar

\3-grams:
-0.01916512     more . </s>
-0.0283603      on a little      -0.4771212
-0.0283603      screening a little     -0.4771212
-0.01660496     a little more    -0.09409451
-0.3488368      <s> looking higher
```

```
-0.3488368      <s> looking on    -0.4771212
-0.1892331      little more loin
-0.04835128     looking on a      -0.4771212
-3      also would consider       -7
-6      <unk> however <unk>       -12
-7      to look a

\4-grams:
-0.009249173    looking on a little    -0.4771212
-0.005464747    on a little more       -0.4771212
-0.005464747    screening a little more
-0.1453306      a little more loin
-0.01552657     <s> looking on a       -0.4771212
-4      also would consider higher     -8

\5-grams:
-0.003061223    <s> looking on a little
-0.001813953    looking on a little more
-0.0432557      on a little more loin
-5      also would consider higher looking

\end\
```

为了使用 test.arpa 评估句子的概率，可以在 kenlm 目录下执行如下命令。

```
import kenlm
model = kenlm.Model('lm/test.arpa')
print(model.score('this is a sentence .', bos = True, eos = True))
```

输出结果：

```
-49.579345703125
```

可以由 C++源代码编译出 KenLM 可执行文件。如果只想要编译出查询代码，可以执行如下命令。

```
./compile_query_only.sh
```

为了编译全部的代码，需要先安装 Boost 库等软件。

```
$sudo apt-get install build-essential libboost-all-dev cmake zlib1g-dev
```

```
libbz2-dev liblzma-dev
```

成功安装 Boost 等软件后，就可以编译 KenLM。

```
$mkdir -p build
$cd build
$cmake ..
$make -j 4
```

使用修改的 Kneser-Ney 平滑法从输入文本 text.arpa 估计语言模型。

```
$bin/lmplz -o 5 <text >text.arpa
```

创建一个三元模型，执行如下命令。

```
./lmplz --order 3 --text input.txt.tok --arpa output.arpa
```

8.3　ARPA 文件格式

统计语言模型描述了文本的概率，它们是在大型文本数据集上训练得到的。可以以各种文本和二进制格式存储统计语言模型，但语言建模工具包支持的通用格式是称为 ARPA 格式的文本格式。此格式非常适合包之间的互操作性。ARPA 格式不如二进制格式有效，因此对于生产而言，最好将 ARPA 格式转换为二进制格式。

用于存储 n-gram 回退语言模型的格式定义如下：

```
<LM_definition> = [ { <comment> } ]
                \data\
                <header>
                <body>
                \end\
   <comment> = { <word> }
```

ARPA 样式的语言模型文件分为两个部分——头部和 n-gram 定义。

头部包含文件内容的描述如下：

```
<header> = { ngram <int>=<int> }
```

上述命令中，第一个<int>给出 n-gram 阶数，第二个<int>给出 n-gram 条目的数量。

例如，bigram 语言模型由 unigram 和 bigram 两个部分组成。头部中的相应条目表示该部分的条目数，条目数可用于辅助装入程序。正文部分包含语言模型的所有部分，定义如下：

```
    <body>  = { <lmpart1> } <lmpart2>
  <lmpart1> = \<int>-grams:
              { <ngramdef1> }
  <lmpart2> = \<int>-grams:
              { <ngramdef2> }
<ngramdef1> = <float> { <word> } <float>
<ngramdef2> = <float> { <word> }
```

考虑文本：

```
wood pittsburgh cindy jean
jean wood
```

在上述文本中，有 pittsburgh, cindy, wood, jean 4 个单词。词汇量大小实际上是 7 个单词，其他 3 个单词是句子开始、句子结尾和未知单词，这些在 KenLM 中分别用<s>、</s>和<unk>表示。这些符号有助于更一致地处理文本。你可以总是用<s>开始一个句子并进一步扩展它。

一个语言模型可能是一个单词序列的列表，列表中的每个序列都有标记在它上面的统计估计语言概率。单词序列可能有、也可能没有与其相关的“回退权重”。在 N 元语言模型中，所有 N-1 元单词序列通常有与它们相关的回退权重。如果某个特定的 N-gram 未列出，则可以从语言模型计算其概率。

P(word_N | word_{N−1}, word_{N−2},···, word_1) =

P(word_N | word_{N−1}, word_{N−2},···, word_2) × backoff-weight(word_{N−1} | word_{N−2},···, word_1)

如果序列“word_ {N−1},word_ {N−2},···,word_1”也未列出，那么术语回退权重“word_ {N−1} | word_ {N−2},···,word_1”用 1.0 替换，递归继续如此。

以下是用 2-gram 语言模型和词汇构建的随机例子，是标准 ARPA 格式语言模型的示例。格式如下：

```
P（N-gram 序列） 序列  BP（N-gram 序列）
```

与下面例子中的 unigrams 和 bigrams 相关的数字是实际概率。因此，如果没有看到序列“wood pittsburgh”，可以通过如下公式来获得它的概率。

$$P(\text{pittsburgh}|\text{wood})=P(\text{pittsburgh}) \cdot BWt(\text{wood})$$

实际概率由其对数替换。通常，对数底数为 10。因此，你将看到的是负数，而不是 0 和 1 之间的数字。

语言模型如下：

```
\data\
ngram 1=7
ngram 2=7

\1-grams:
-1.0000 <unk>    -0.2553
-98.9366 <s>     -0.3064
-1.0000 </s> 0.0000
-0.6990 wood     -0.2553
-0.6990 cindy    -0.2553
-0.6990 pittsburgh        -0.2553
-0.6990 jean -0.1973

\2-grams:
-0.2553 <unk> wood
-0.2553 <s> <unk>
-0.2553 wood pittsburgh
-0.2553 cindy jean
-0.2553 pittsburgh cindy
-0.5563 jean </s>
-0.5563 jean wood

\end\
```

ARPA 模型可以包括三元，甚至多元。七元和十元是罕见的，但仍然有人使用。使用 gzip 可以压缩 ARPA 模型以节省空间。

8.4 依存语言模型

使用依存语言模型可以创建单词之间的句法依赖关系模型。下面讲解使用依存文法构建依存语言模型。

依存文法认为，词之间的关系是有方向的，通常是一个词支配另一个词，这种支配与被支配的关系就称为依存关系。包括汉语和英语的大多数语言均满足投射性。所谓投射性，是指如果词 p 依存于词 q，那么 p 和 q 之间的任意词 r 就不能依存到 p 和 q 所构成的跨度之外。汉语句子“这是一本书。”的依存文法结构如图 8-6 所示。

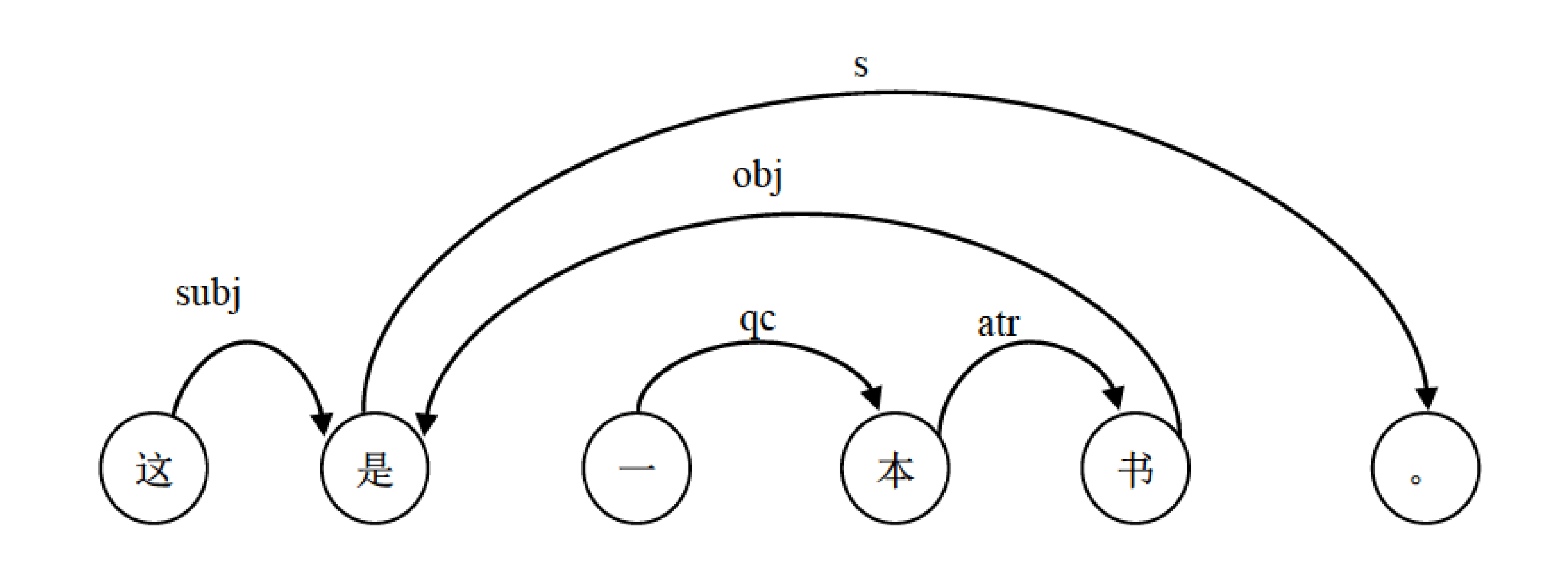


图 8-6 “这是一本书。”的依存文法结构

图 8-6 中带箭头弧的起点为从属词，箭头指向的为支配词，弧上标记为依存关系标记。例如，句号“。”支配“是”，动词“是”为句子的谓语，它支配主语“这”和宾语“书”，则“是”为支配词，“这”和“书”为从属词，“s”“subj”“obj”为依存关系标记。支配词也称为核心词，从属词也称为修饰词。

再如，数词“一”作量词“本”的量词补足语，“本”为支配词，“一”为从属词，“qc”为依存关系标记。数量短语“一本”作名词“书”的定语，名词“书”支配量词“本”，“atr”为依存关系标记。

依存文法也可以表示成图 8-7 这样的树形结构。因为总是连接线下面的词依赖上面的词，所以图 8-7 中的箭头可以省略。

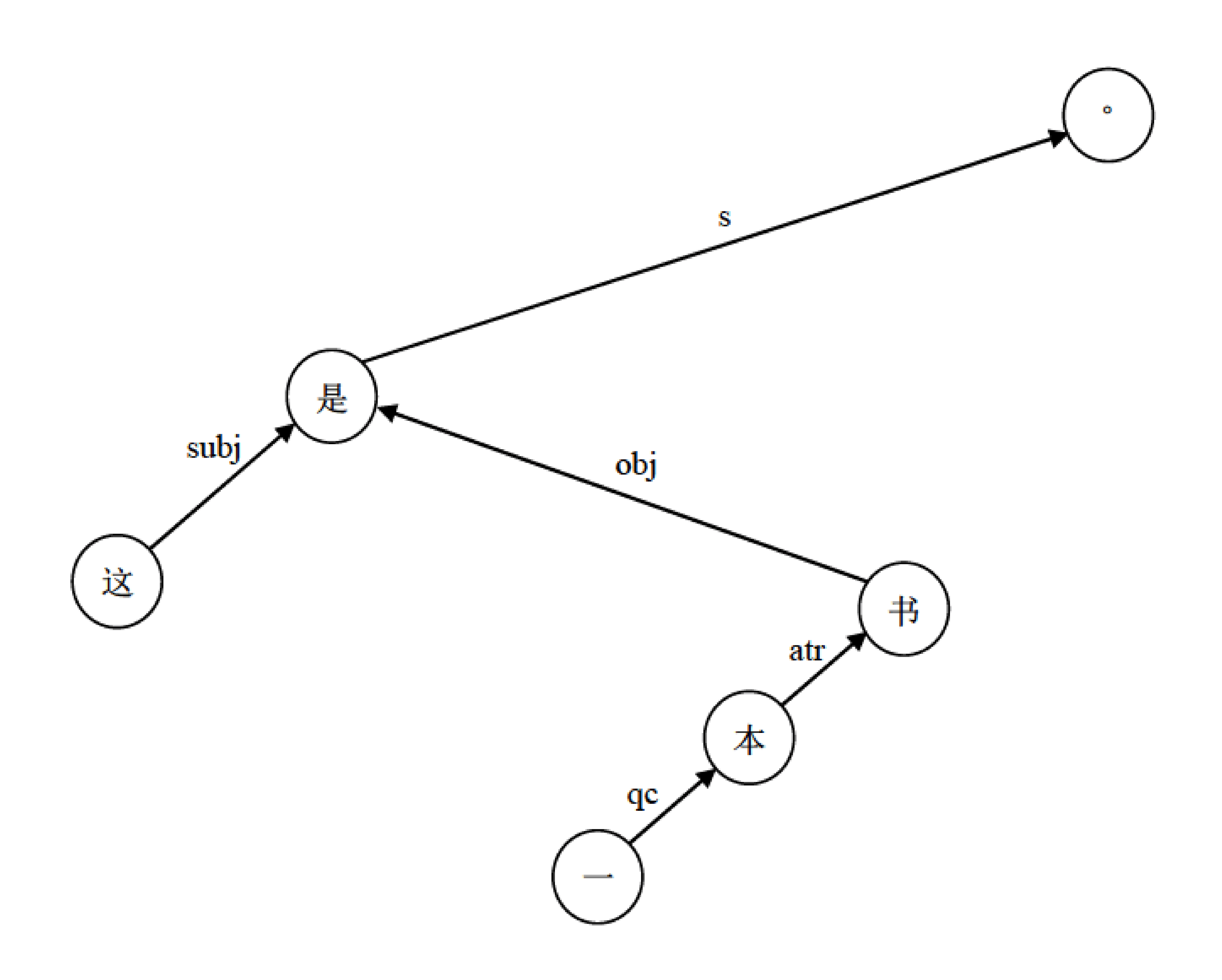


图 8-7 “这是一本书。”的依存文法树

在依存语言模型中利用依存树有很多可能的方法。构造依存语言模型的最简单方法是使用依存树的拓扑结构 T。每个单词都由其父结点调节。图 8-8 中句子的概率计算公式如下：

$$P(s \mid T) = P(\text{the} \mid \text{boy})\, P(\text{boy} \mid \text{find})\, P(\text{will} \mid \text{find})\, P(\text{find} \mid \text{<NONE>})\, P(\text{it} \mid \text{find})\, P(\text{interesting} \mid \text{find})$$

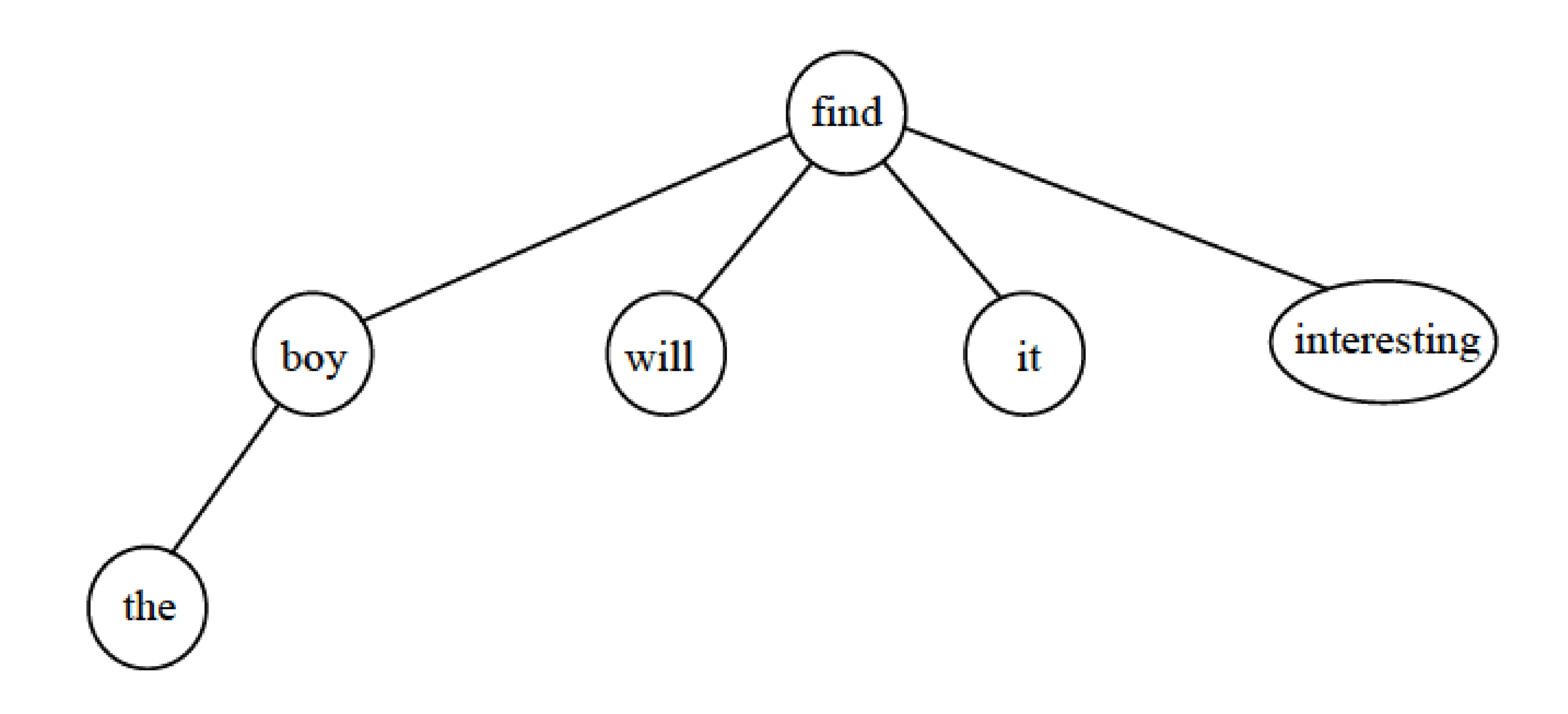


图 8-8 句子“the boy will find it interesting”的依存树

附　录

术语及含义

术　语	含　义
LVCSR	全称为 Large Vocabulary Continuous Speech Recognition，大词表连续语音识别
Automatic speech recognition	自动语音识别
Endpoint Detection	端点检测
Phone	音素
Probability Density Function	概率密度函数
Speech Synthesis	语音合成
Subspace Gaussian Mixture Model	子空间高斯混合模型
Semiring	半环在抽象代数中，半环是与环相似的代数结构，但不要求每个元素必须具有加法可逆
Triphone	三音素
Acoustic Model	声学模型